Master's Thesis

Towards x86 Instruction Set Emulation in Java via Project-based Text-to-Code Generation using Reinforcement Learning

2210422 TRAN, Thu Thi Anh

Supervisor	Prof. Mizuhito Ogawa
Main Examiner	Prof. Mizuhito Ogawa
Examiners	Prof. Nguyen Le Minh
	Prof. Kiyofumi Tanaka
	Prof. Naoya Inoue

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2024

Abstract

Malware analysis by formal methods using Control Flow Graphs (CFGs) has been proved to be more effective than conventional signature-based strategies. To reconstruct a CFG of a given program, Dynamic Symbolic Execution (DSE) techniques are often used. The implementation of a DSE tool must strictly comply with the specifications of its designated architecture - the Instruction Set Architecture (ISA) manual. As there is a number of computer processor families with each has several variations and editions, fully manual DSE tools construction certainly demands extensive engineering work. To help reduce human effort, tasks such as environment emulation and instruction set emulation can be semi/fully automated with the help of natural language processing techniques.

The semi-automated approach of such tasks includes two steps: extracting semantics from natural language text of the ISA manual and mapping them into a prepared code template tailored to the platform that constructs the DSE tool. Two notable DSE tools which are BEPUM (Binary Emulation for PUshdown Model) for x86 architecture and CORANA for ARM architecture employs semi-automatic instruction set emulation. It is reported that BE-PUM successfully generates Java code implementation for 56.41% of 530 selected x86 instructions and CORANA scores at 63.72% of 1039 ARM - Cortex M instructions in 5 variations. While achieving promising emulation results, this approach still requires manual preparation of both interpretation rules for semantic extraction and project-based code templates. Although the current progress of BE-PUM and CORANA shows that the amount of human effort spent on the manual preparation is minimal compared to the traditional workload, it is evidence that to yield higher results than those does demand greater human labor.

The fully automated approach eliminates the need for rule preparation, concentrating instead on end-to-end text-to-code generation. In this study, we explore the feasibility of this approach by developing CoDeb system which aims at applying reinforcement learning to large language models for fully-automatic emulation of x86 instruction set based on its description in natural language, utilizing feedback from compiler and the existing Java codebase of BE-PUM project. As a result, the performance of this method would not be bounded by human effort. However, the quality of the automatically generated codes must meet standard requirements, including syntactical and semantic correctness.

The scope of our study focuses on ensuring project-level syntactical correctness via successful compilation. This requires that the generated code is valid within the project-level context of BE-PUM, meaning it must correctly utilize the existing code base, including function calls, variable names, and data types. In our work, we adopt two generative models, one acts as a code writer (Coder) and the other as a code debugger (Debugger), hence the name CoDeb. Additionally, the code base knowledge of BE-PUM project is built into separate vector database which serves as syntax references for the generative models. To eliminate the need for manual work spent on preparing labelled dataset or coding examples, we approach via almost-zero-shot generation by preparing a small code template and employing a set of rule-based feedback and compiler feedback to help iteratively improve the generation through reinforcement learning with Proximal Policy Optimization. Out of 200 selected x86 instructions, CoDeb's best attempt successfully generates project-level compilable

code for 20 instructions, achieving a 10% success rate. Due to time and computing resource constraints, only this attempt (among other experimental trials) completed a total of 1,147 instructions, achieving a 14.39% success rate with 165 successfully compiled instructions. Compared to the baseline of semi-automatic approaches, our work, though with modest results, shows promising potential for application and adaptability.

Keywords — Project-level Text-to-Code Generation, x86 Instruction Set Emulation, Reinforcement Learning, Rule-based Feedback, Compiler Feedback, Code Writer, Code Debugger, Knowledge base.

Acknowledgement

First and foremost, I would like to express my heartfelt gratitude to Professor Mizuhito Ogawa for his exceptional guidance and support throughout my master years at Japan Advanced Institute of Science and Technology. His deep knowledge, thoughtful advice, and constructive feedback have significantly shaped the direction of my work. I am truly grateful for the time he has invested in helping me navigate challenges. Working under his supervision has been a rewarding experience; not only has he inspired me to pursue further academic career, but he has also shown me how to be better at approaching it, from thinking critically to tackling problems. I deeply appreciate his contributions to my growth as a scholar.

I wish to express my sincere thanks to my second supervisor, Professor Naoya Inoue, for his insightful advice and thoughtful guidance on the theme of my work. His insights have been instrumental in shaping and refining my research.

I would like to extend special thanks to my minor research supervisor, Professor Nguyen Le Minh, for his guidance on the general theme of natural language processing. His insights into the current state of the field have been incredibly valuable.

I would especially like to express my appreciation to my seniors and friends. My special thanks go to my lab members, Mrs. Nguyen Thi Van Anh, Mr. Pham Thanh Hung, Ms. Nguyen Thi Hai Yen, Mr. Kosuke Udatsu, and Mr. Nguyen The Hung for the wonderful times we shared in both our studies and daily life. In particular, Mrs. Nguyen Thi Van Anh has consistently provided me with invaluable help and advice. I am also grateful to my group of friends who have shared memorable experiences with me when we went scaling mountains together.

Last but not least, my family is my greatest source of motivation. I deeply thank them for always standing by my side and encouraging me to keep moving forward.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The central focus of our study is to apply reinforcement learning to large language models for fully automated project-level text-to-code generation, utilizing compiler feedback and an existing project's code base. This research stems from the essential need to minimize human effort in developing large-scale projects for malware analysis tools. This section outlines our motivation, beginning with the demand for automation in malware analysis and leading to the rationale behind our proposal.

Malware (malicious software) is a broad term for software created by cyber-criminals to exploit computer system's vulnerabilities, either to damage the system or gain unauthorized access to its data. Malware includes several types, such as viruses, spyware, trojans, and ransomware. These threats can lead to severe consequences, including data breaches and system failures. One of the largest and most impactful malware incidents is the WannaCry ransomware attack of May 2017 [1]. This global cyber-attack affected over 200,000 computers in 150 countries. WannaCry encrypted files on affected Microsoft Windows computers, and demands Bitcoin ransoms for decryption keys. It disrupted both private users and major organizations including hospitals, schools and government establishments. The incident, along with vast amount of cyber-attacks over the past years, poses a dire need for early prevention methods against malware.

Malware analysis is one of the preemptive prevention techniques that includes malware classification and detection. Surveys [1, 2] provides an overview of various malware analysis techniques starting from conventional static analysis with non-executed code examination, to dynamic, hybrid analysis and most recently, machine learning methods. With the evolution of malware, static analysis such as signature-based anti-virus tools has now fallen behind because techniques like code obfuscation and encryption can by-pass static checking. Meanwhile, dynamic analysis and its hybrid version are more preferred. By analysing through the dynamic execution of malware in contained and controlled environments, they are more robust and accurate compared to the traditional solutions. One of the dynamic analysis tools that is developed in our laboratory is BE-PUM (Binary Emulation for PUshdown Model) [3]. BE-PUM is a binary analyzer designed for x86 malware. It employs Dynamic Symbolic Execu-

[1] https://www.malwarebytes.com/wannacry

tion (DSE) to reconstruct the malware's precise Control Flow Graph (CFG), which can be then used to accurately trace software's behaviors.

Since a DSE tool must be tailored to its target computer processor architecture such as ARM, x86, and MIPS, being a symbolic execution tool for x86 architecture, BE-PUM strictly requires the emulation of the processor's instruction set in order to explore execution paths and construct the CFGs. The emulation of the x86 instructions involves describing their semantics in Java - the main programming language that is used in the development of BE-PUM project. There are over a thousand x86 instructions and their variants. Although working with malware involves non-floating-point instructions, manually implementing these instructions still requires a substantial amount of human effort and resources. The typical implement process comprises parsing through the Instruction Set Architecture (ISA) manual to gasp the conceptual model of processor's environment, rules and operations. Then from such description in natural language, via the programming language used in the DSE development project such as Java in our case, the human developer describes the semantics of the instructions, organizes the code files, and builds tests to verify the implementation's correctness. Hence, we can see that emulating the instruction set alone imposes a significant overhead and consumes considerable time that could otherwise be spent on the main purpose which is to develop a DSE tool.

Nowadays, with the rapid progress of automatic text-to-code generation techniques, it is reasonable to dedicate the emulation stage to these methods. Along with the history of natural language processing evolution, there are two major approaches: semi-automatic and fully end-to-end automatic text-to-code generation.

Semi-automatic text-to-code generation. The previous code base of BE-PUM [4] has already included semi-automatic instruction set emulation in which the authors devise grammar rules of the x86 instruction set's pseudo-code and map them to the grammar of Java, which is eventually used to generate Java codes. Additionally, sentence similarity measurement through TF-IDF (Term Frequency - Inverse Document Frequency) is used to assess flag update cases. It is reported that this method successfully generates Java code implementation for 56.41% of 530 selected x86 instructions. Another notable DSE project employing a semi-automatic method is CORANA, designed for the ARM (Advanced RISC Machine) architecture [5]. The authors extract semantics of ARM instructions from natural language text in the ISA manual via syntax parsing and template mapping. The code templates are manually prepared and tailored to the code project that constructs the DSE tool. The solution is able to cover 63.72% of 1039 ARM - Cortex M instructions in 5 variations. While achieving promising emulation results, this approach still requires manual preparation of 35 initial functions and 228 rewrite rules. In conclusion, the current progress of both BE-PUM and CORANA shows that the amount of human effort spent on the manual preparation is indeed minimal compared to the traditional workload. However, it is evident that to achieve better results would require significantly more human effort.

Fully-automatic text-to-code generation. Different from the aforementioned approach, the fully automated approach removes intermediate rule preparation and directly transforms the natural text input into programmable codes. The backbone of these methods is often machine learning models. Text-to-code generation is claimed to obtain impressive results with the current state of large language models [6], however, such results are only of function-level generation or file-level generation where all variables and methods are self-contained within the said function or file. In practice,

software development comes with large and complex code base such as which of BE-PUM, it thus renders function-level or file-level generation insufficient. Additionally, to enable project-level generation, the context of the code base should be supplied to the generative model, through a knowledge base for instance. Unlike text-to-code models developed through supervised learning, models that depend on feedback from standard software programming verification methods like compilation and testing are more advantageous. This is because they eliminate the need for labeled datasets, which are often labor-intensive to create. Consequently, these models can be framed as reinforcement learning problems.

Our target. Given the circumstances, we pursue the latter approach - fully-automatic project-based text-to-code generation for the emulation of x86 instruction set in BE-PUM. Upholding the goal of reducing human labor, we utilize automatic verification from one of the software programming processes which is code compilation, hence, reinforcement learning on large language models is needed. In a bigger picture, our theme is to generate executable codes, especially on project-level, from a given well-defined software specification in natural language.

1.2 Problem Statement

The ISA manuals are technical documents that describe the architecture and behavior of a computer's instruction set. Different from casual text, the language used in these manuals has distinct characteristics to ensure clarity, precision, and unambiguity. These characteristics include:

1. Self-contained knowledge: Targeting at engineers, researchers and students, ISA manuals provide all the necessary information to understand and work with a computer's instruction set independently.
2. Technical precision: ISA manuals use precise definitions and formal language to clearly describe each instruction's syntax and semantics and thus avoids ambiguity.
3. Highly-hierarchical structure: ISA manuals organize information into several levels of details and categories. As such, this leads to frequent cross-references between sections.
4. Instructional sentences: Especially for the instruction description, each sentence concretely describes a step of the operation. Pseudo-codes are optionally provided.

Property #1 makes automatic emulation of instruction set possible as its specification can be sufficiently retrieved from a single source. Properties # 2 and #4 makes aforementioned semi-automatic approaches feasible and thus, it is promising for fully-automatic methods. However, property #3 makes it hard for retrieving natural language description into a whole text body. Additionally, for property #4, it is not always the case that pseudo-codes are guaranteed to be included in every ISA manual. Despite the natural language descriptions, having a pseudo-code section is more advantageous as it closely resembles actual code implementation. However, pseudo-code of each instruction is not self-contained because to enable code re-usability, a group of steps may be packed into a separate function. These functions are often documented in other sections of the manual or may not be documented at all as

pointed out by [4] for x86 case. Therefore, generating instruction's code implementation should not be relied solely on the availability of pseudo-codes. As such, a natural language text-to-code solution has higher degree of generalization and hence, is more preferred.

Our project-based text-to-code generation problem is stated as follows:

- Inputs:
  - Let $\mathbf{x} \in \mathcal{D}$ is the description of each x86 instruction in natural language which includes a text sequence describing its operation and flag update. $\mathcal{D}$ is the finite set consists of selected instructions.
  - Let $\mathbf{x}' \in \mathcal{T}_{\text{BE-PUM}}$ is the text sequence of a prepared code template tailored to the code context of project BE-PUM. The template is used with two purposes: 1) To give sample code snippets on retrieving program's variables and 2) To be a main frame of a code file that is to be completed by the system. With the template be instantiated for each instruction by simple string substitution $\mathbf{x}' = \text{MakeTemplate}(\mathbf{x})$, $\mathcal{T}_{\text{BE-PUM}}$ is the finite set of all involved templates.
- Output: Let $\mathbf{y} \in \mathcal{Y}$ be the token sequence for the content generated by the system, $\mathcal{Y}$ is finite. The desired characteristic of $\mathbf{y}$ is that it is successfully compiled, in another word, it should pass Java compiler (Javac) without any errors.
- Hence, our system CoDeb is as follows:

  $$\mathbf{y} = \texttt{CoDeb}(\mathbf{x}, \mathbf{x}', T_{\text{init}}, T_{\text{iter}})$$

  where T_{init} defines the maximum number of times the system can attempt to write code and T_{iter} defines the maximum number of debugging iterations that the system is allowed to fix its generated codes.

Taking an inspiration from *Pair Programming* — a collaborative practice where two developers work together on the same code, CoDeb system employs two large language models for the positions of Code Writer and Code Debugger. The Code Writer is responsible for code creation, while the Code Debugger focuses on identifying and fixing errors. The flow of components within the system resembles how the two programmers having a conversation on writing codes, as illustrated in Figure 1.1. The figure is a simplification of the CoDeb system, highlighting two generative models exchanging messages via natural language sequences. The main flow of CoDeb is as follows:

- The first code generation of the Code Writer is called Initial Code Generation. This stage allows the Code Writer to attempt at producing a complete Java code file for the given input within a number of trials.
- The Java code file is then passed into a Reward Function to obtain feedback from multiple checking stages. The result is called Compiler feedback, consisting a scalar reward score and compilation messages.
- The Code Debugger looks into the compilation error in the feedback and the code written by the Code Writer to provide guidance for code correction. This includes an explanation and a suggestion drawn from knowledge base BEPUM-KB to fix the error.

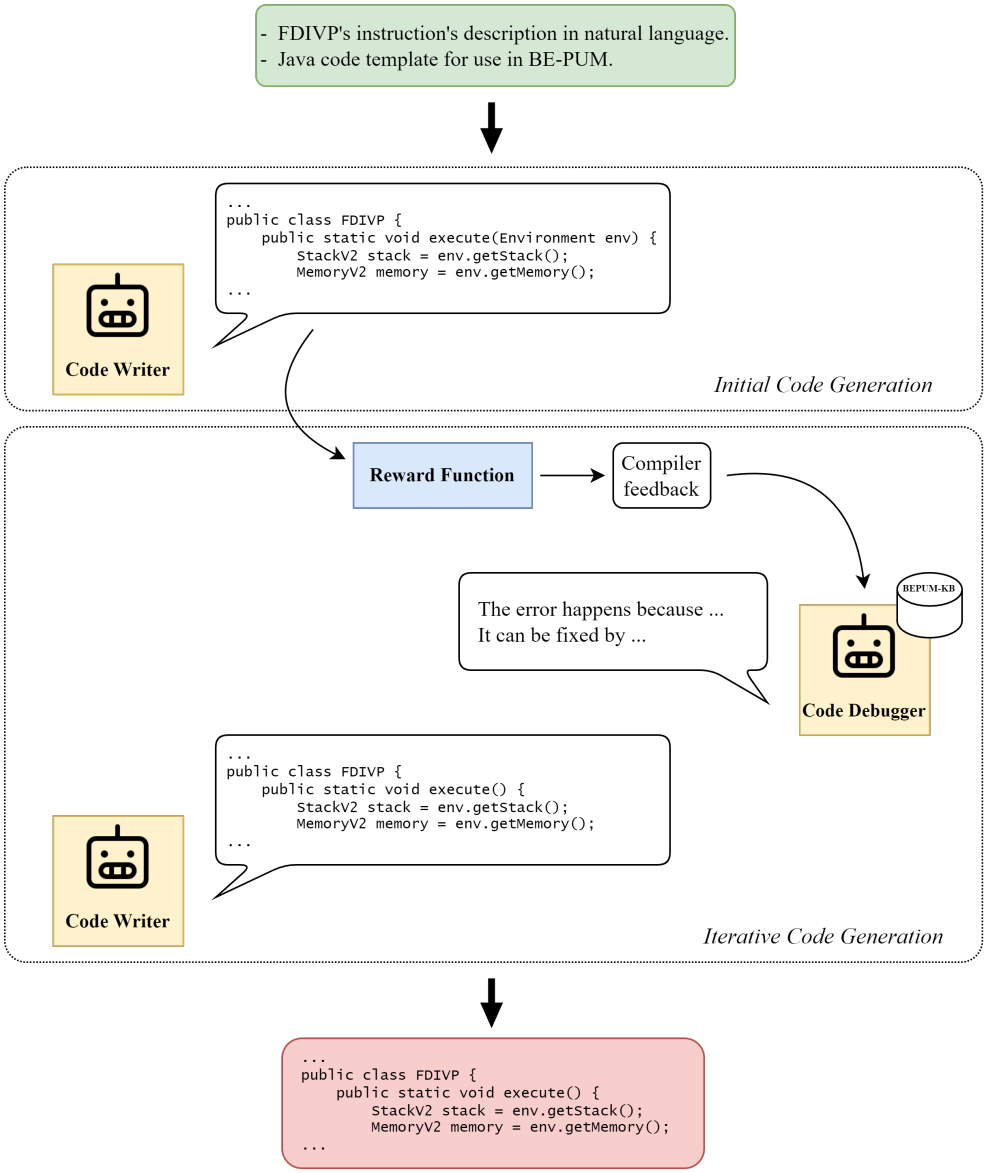


Figure 1.1: Simplified overview of our CoDeb system in terms of message communication for producing code implementation for instruction FDIVP.

- Receiving the guidance from the Code Debugger, the Code Writer produces another Java code file which is its attempt to fix the said error. This starts a new iteration in the Iterative Code Generation stage.

When any of the terminating conditions is met, either the Java code file is compilable in BE-PUM project or the Code Writer uses up all of the allowed iterations, the current answer of the Code Writer is the final output of the system CoDeb.

The above explanation presents the flow of operation. For the large language model slots, currently we use *CodeLlama-7b-Instruct-hf*. For the construction of the system, reinforcement learning is involved in fine-tuning the two generative components. We choose Proximal Policy Optimization (PPO) to optimize them. This learning scheme makes use of the numerical value of the Compiler feedback. Additionally, typical fine-tuning techniques for large language models are also employed, including b-bit quantization [7] and Low-rank Adaptation (LoRA) [8]. The typical workflow of reinforcement fine-tuning generative models often includes a supervised fine-tuning (SFT) step which requires an amount of labeled data, to govern the output format. As our work aims at adhering to practical scenarios of programming where there is no sample codes provided beforehand, we instead remove SFT step by crafting a short code template and incorporating the checking on output format as criteria used in the Reward Function. Hence, our problem can be framed as an almost zero-shot text-to-code generation as no sample program is demonstrated but rather just the core frame of the code file.

Difficulties. As an initial attempt to tackle instruction emulation using generative large language models, CoDeb faces the challenge of constrained decoding [9]. In addition to that, being a stateless system where past code correction activities are not retained, the correct output of CoDeb is not progressive kept throughout iterations. Hence, its results are lower compared to which of the semi-automatic approaches. Other factors, such as time constraints and limited computing resources, also contribute to the situation.

Achievements. We experiment CoDeb with several different configurations. Given the situation and said difficulties, CoDeb's best effort successfully translates 20 out of 200 selected x86 instructions into project-compilable code, yielding a 10% success rate. Due to the limitation of time and resources, only one experiment, which is the above best-effort case, is able to complete a full dataset of 1147 x86 instruction. In such case, CoDeb produces 165 compilable codes, resulting in a 14.39% success rate. Compared to traditional semi-automatic methods, these results present the potential and flexibility of our approach.

1.3 Related Work

There have been numerous research on this topic achieving state-of-the-art results with respect to standard benchmark (e.g: CONALA [10], CodeXGLUE [11], APPS [12]) for code generation. For instance, the closest work that shares the same idea of using compiler feedback to perform reinforcement fine-tuning code generation is Bi et al. [13]. The work focuses on Python code generation from natural language description. To build the iterative code refinement process, the authors prepare a SQL vector database of error messages from compiler and the corresponding project context (code fragments) related to such errors. The limitation of the solution includes

two major points: 1) The preparation of possible compilation errors may not cover all cases and hence may not generalize well to unseen cases in practice; 2) To query into the SQL database of pre-defined errors, the query string must be generated in runtime by ChatGPT-v2 which is a commercial model.

Another work that shares the idea of iterative code generation using two trainable models is CodeRL [14]. CodeRL is a framework for program synthesis that uses pre-trained language models and deep reinforcement learning. More specifically, the code-generating language model is treated as an actor network, while the other is used as a critic network to evaluate the functional correctness of the generated programs and provide *dense* feedback signals to the actor. Different from our approach, the second model other than the code generator which is the critic model, is trained through supervised learning as an error predictor rather than a code debugger as ours. In training time, the critic model receives problem definitions and ground-truth programs to learn to predict one of four possible unit tests' outcomes: {`CompileError, RuntimeError, FailedTest, PassedTest`}. The probability of a specific unit test outcome from the four possible ones is then used to influence the training of the actor model. As probability value is used as feedback for the actor, it is called *dense* feedback signal.

The literature provides strong evidence supporting the application of reinforcement learning for code generation using feedback from software engineering tool verification. It also emphasizes the great concern of producing code programs that are complied to project-level context. Furthermore, it is suggested that iterative improvements are beneficial for refining the output of code generation programs.

1.4 Contribution

The main contribution of our work includes:

- We investigate the potential of employing two generative large language models conversing in natural language to write and improve code together. Work in the literature of text-to-code generation often tailor the debugging stage with labeled data. Our work instead grants it to a generative model which, by its pre-trained knowledge, is capable of code debugging. Evidently, our experiments have yielded notable results.
- We contribute to the theme of project-based or project-level code generation by supplying generative components with project context, represented as code lines that are potentially relevant or similar to the line of code that needs correction. The process of building the knowledge base of project code lines also involves those that are explored from traversing project's class diagram. These code lines may not have been existed beforehand in the current code base, which does not limit the correction to existing implementations.
- Our work shows that it is possible to by-pass SFT step for reinforcement learning with generative language models and instead enforcing the format rules through reward function.
- Our CoDeb system represents an initial attempt to leverage generative models for fully-automatic end-to-end implementation of x86 instructions based on their

English descriptions. We discuss its advantages and limitations and outline potential future directions.

1.5 Thesis Outline

The thesis is composed of 8 chapters. Chapter 1 is the introduction to our research theme. The remaining 7 chapters are as follows:

- **Chaper 2** presents our task-specific domain, including knowledge on x86 architecture and the current state of BE-PUM.
- **Chaper 3** overviews the foundations on language models as well as their typical fine-tuning techniques that are employed in our work.
- **Chaper 4** briefly gives foundations on reinforcement learning which is the technique used in the fine-tuning of the generative components. This section shows the feasibility of using software programming verification for text-to-code generation.
- **Chaper 5** is dedicated to explain our proposed system CoDeb in depth. It details each component of the system, their functions and interactions, as well as the reasoning behind our system design.
- **Chaper 6** reports the results of the CoDeb system in multiple settings. The results include both statistic data and figurative demonstration data.
- **Chaper 7** discusses the achievements and drawbacks of our approach.
- **Chaper 8** concludes the thesis and proposes future plan for our work.

Chapter 2

X86 Architecture and BE-PUM

2.1 X86 Architecture

The x86 architecture, developed by Intel, is a CISC (Complex Instruction Set Computing) architecture widely used for central processing units (CPUs) in desktop, laptop, and server computers. The name "x86" originally refers to the entire family of backward-compatible processors whose model numbers end in "86" such as the 80186, 80286, 80386, and 80486. Starting from model 80386, "x86" is used synonymously with the 32-bit version (the IA-32 – Intel Architecture, 32-bit) that supports 32-bit wide data paths, registers, and memory addresses. A typical x86 ISA defines 4 key components of the architecture including the basic execution environment (memory model and registers), the instruction set specifications, the handling of interrupts and exceptions, and backward compatibility. This section is dedicated to overview the key components with a focus on the instruction set specifications.

2.1.1 X86 Basic Execution Environment

The basic execution environment can be further divided into 4 elements: Register, Flag, Memory and Stack.

Register

In computer architecture, a register is a small amount of storage available directly within the central processing unit (CPU). Registers are used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU. They are much faster than the main memory (RAM) and are essential for the CPU's operations. The x86 architecture includes several types of registers, each serving specific purposes. The standard bit length for an x86 register is 32 bits. However, shorter bit lengths are also used when necessary. To accommodate this, a 32-bit register can be divided into halves or quarters as shown in Figure 2.1. Based on functionality, x86 registers can be categorized as follows:

- General-purpose registers: These registers can be used for a variety of tasks and are often involved in arithmetic and data manipulation. There are 8 32-bit registers along with 8 16-bit sub parts and 8 8-bit sub parts in this category (Figure 2.1).

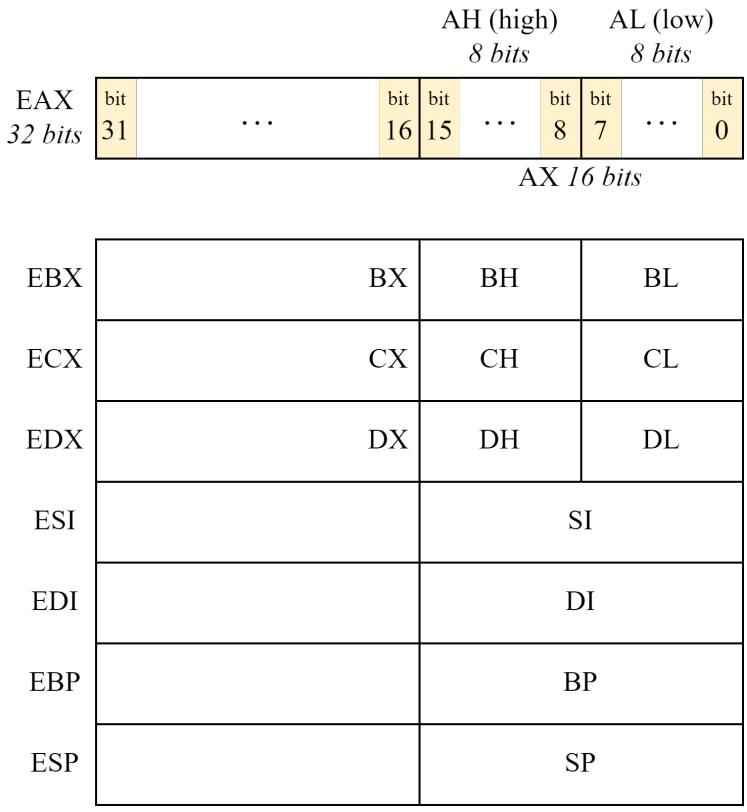


Figure 2.1: x86 General-purpose registers with bit lengths.

- Segment registers: These registers hold segment selectors, which are used to access different memory segments (E.g: CS, DS, SS, ES, FS, GS).
- Instruction pointer register: It holds the address of the next instruction to be executed (E.g: EIP, IP).
- Flags register: The EFLAGS register is a 32-bit register where each bit represents a binary value. The details is describe in component Flag below.

Flag

Status flags, control flags, and system flags that indicate the results of operations and manage the CPU's behavior are stored on a 32-bit EFLAGS register. The key flags include:

- CF (Carry Flag) - *1 bit*: Indicates an overflow in arithmetic operations.
- ZF (Zero Flag) - *1 bit*: Indicates whether the result of an operation is zero.
- SF (Sign Flag) - *1 bit*: Indicates the sign of the result of an operation.
- OF (Overflow Flag) - *1 bit*: Indicates an overflow in signed arithmetic operations.
- PF (Parity Flag) - *1 bit*: Indicates if the number of set bits in the result is even or odd.
- AF (Auxiliary Carry Flag) - *1 bit*: Used in binary-coded decimal (BCD) arithmetic operations.

- DF (Direction Flag) - *1 bit*: Controls the direction of string operations.
- IF (Interrupt Flag) - *1 bit*: Controls the enabling and disabling of interrupts.

The full list of flag bits and their positions on the EFLAGS register is shown in Figure 2.2[1] with several bits are reserved or have constant values throughout CPU execution.

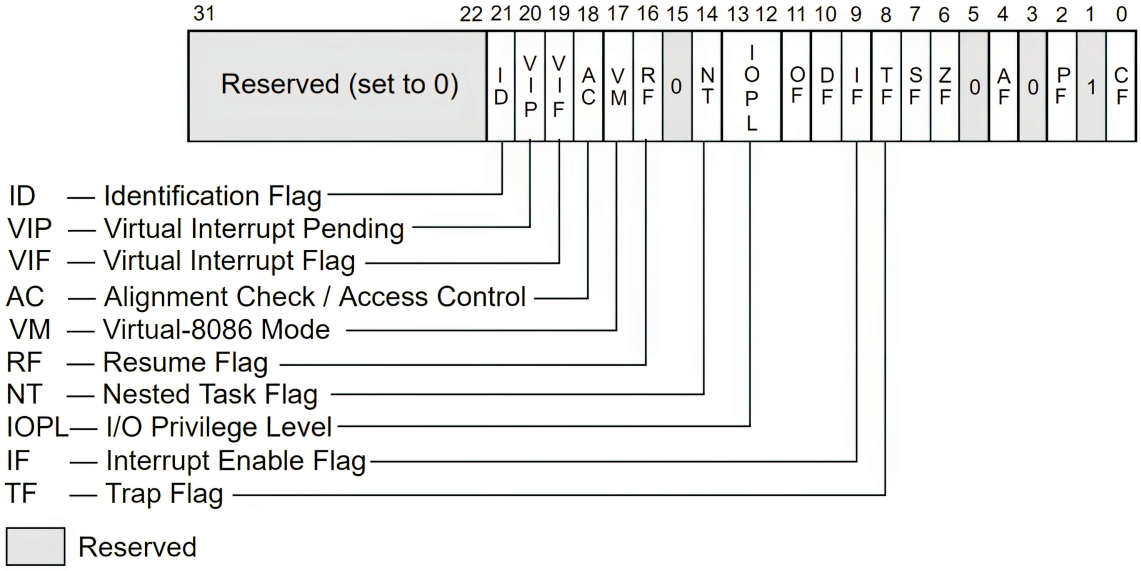


Figure 2.2: EFLAGS Register.

Memory

The x86 memory model and management handles memory access, organization, and protection. X86 provides three types of memory models as follows:

- Flat Memory Model: all addresses are treated as a single contiguous block, which allows applications to access a linear address space without segmentation.
- Segmentation: The memory is divided into different segments (code, data, stack). Each segment has a base address and a limit.
- Paging: The virtual address space is divided into fixed-size blocks called pages (typically 4 KB). The operating system uses a page table to map virtual addresses to physical addresses, which is then used for memory access.

The byte order scheme of x86 architecture is primarily little-endian. This means that in multi-byte data types (like integers or floating-point numbers), the least significant byte is stored at the lowest memory address, and the most significant byte is stored at the highest address. The x86 registers' bit array also follows such manner as shown in Figure 2.1.

Stack

The stack is a region of memory used for temporary storage of data, particularly during function calls, local variable storage, and managing control flow. It operates on a last-in, first-out (LIFO) principle - the last item pushed onto the stack will be

[1]The figure is taken from Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A [15].

the first one to pop off next. The stack pointer register (ESP/SP) points to the top of the stack. It is automatically adjusted during push and pop operations.

Each function call creates a stack frame, which includes space for local variables, the return address, and saved registers. The base pointer (EBP/BP) is used to reference the base of the current stack frame.

2.1.2 X86 Instruction Set and Its Specifications

X86 Instruction Set

In general, a computer instruction is a binary-coded operation that a CPU can execute. It tells the processor to perform a specific task, such as arithmetic calculations, or data movement. Because the instruction set is the lowest-level command that controls the CPU, emulating it is crucial for software analysis using formal methods. The instruction set can be categorized into 4 main groups: Data transfer instructions, Arithmetic instructions, Logical instructions and Control transfer instructions.

- Data transfer instructions: Transfer data between registers, memory, and I/O devices (e.g., MOV, PUSH, POP).
- Arithmetic instructions: Perform mathematical calculations (e.g., ADD, SUB, MUL, DIV).
- Logical instructions: Execute bitwise operations (e.g., AND, OR, XOR, NOT).
- Control transfer instructions: Alter the sequence of instruction execution (e.g., JMP, CALL, RET).

The specifications

Regardless of architecture, the ISA manual always reserves a section to describe its exhaust list of the instruction set. Each entry of the list includes specifications of each instruction, organized in a systematic format. The specifications for an x86 instruction includes:

- Encoding scheme:
  - Opcode: A binary sequence that is unique for each instruction, specifying the operation which is recognized by the CPU. If an instruction has several variants, their opcodes are also unique.
  - Operands: (Optionally) At max 4 operands. Each operand can be data or resources such as registers.
- Description: Operation described in natural language. This section is often long, giving information on the typical operation of the instruction along with exceptions and special cases. It may also include specifications on the update of flags, for example, in ARM manual. However, for x86 ISA manual, the flag update section is separated.
- Flag update: Changes of flag bits after executing the described instruction. This section is often short, consisting of one or two sentences.
- Operation: Pseudo-code expressing the operation. The grammar used in writing the pseudo-codes is basic and often not specified explicitly in the manual. Noted

that pseudo-code section is not always available for other architecture families such as ARM - Cortex M series [5].

An excerpt of specification for instruction named "AAM" from Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A [16] is provided in tabular form (Table 2.1). It can be seen that after information about Flag update, there is additional information on Exception, which is not typical for all of the instructions.

Usage of the Instruction Set Manual

An instruction set manual is a critical resource for anyone involved in low-level programming, system development, performance optimization and security analysis. Some of the practical use cases are:

- Operating system development: The ISA manual provides crucial information about the CPU's capabilities, instructions, and behaviors that are essential for building efficient OS. These information includes memory management, system calls and interrupt handling, task management and context switching, I/O operations, debugging and diagnostics.
- Compiler development: The ISA manual provides information on the binary encoding of each instruction, which serves as the reference for selection and generation of machine code from intermediate representations. The specifications on the execution environment also affects the interpretation of high-level data structures into low-level resources.
- Malware analysis: The ISA manual provides detailed information on the CPU's instructions and their behaviors. Malware analysis can benefit in multiple ways:
  - Reverse engineering: Dissemblers can be built using binary encoding of each instruction.
  - Behavior analysis: Emulators, which are software, are able to replicate hardware functionality based on the target architecture specifications. They allow examining the execution of malware without the need to execute them in real environments.

Among the aforementioned use cases, we focus on the instruction set's role in malware analysis, specifically emulating the instruction set used in a malware analysis system. The next section describes our target binary analysis system.

2.2 BE-PUM

There are two major scenarios in binary analysis: analysing system software and analysing malware, with analysing malware involves dealing with tricky obfuscation codes. One of the effective analysis methods is to use dynamic symbolic execution. In order to build such tool, the target architecture's instruction set needs to be emulated. The emulation of the instruction set - describing each instruction's semantics in certain programming language, is costly to manually implement. There are several tools for binary code analysis such as *McVeto* [17], X-Force [18] and BE-PUM [3]. Among which BE-PUM intends its instruction set emulation to be done semi-automatically or fully-automatically. Given access to its code base and its ongoing development, we choose BE-PUM as our target.

Table 2.1: Information of x86 instruction AAM.

AAM — ASCII Adjust AX After Multiply					
Opcode	**Instruction**	**Op/En**	**64-bit Mode**	**Compat/ Leg Mode**	**Description**
D4 0A	AAM	ZO	Invalid	Valid	ASCII adjust AX after multiply.
D4 ib	AAM imm8	ZO	Invalid	Valid	Adjust AX after multiply to number base imm8.
Instruction Operand Encoding					
Op/En	**Operand 1**	**Operand 2**	**Operand 3**	**Operand 4**	
ZO	N/A	N/A	N/A	N/A	

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the imm8 byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 imm8).

This instruction executes as described in compatibility mode and legacy mode.
It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL := AL;
        AH := tempAL / imm8; (imm8 is set to 0AH for the AAM mnemonic)
        AL := tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#DE	If an immediate value of 0 is used.
#UD	If the LOCK prefix is used.

BE-PUM (Binary Emulation for Pushdown Model) [3] is a binary file analyzer for Intel x86/Win32 malware that employs DSE technique to generate precise control flow graphs (CFG) in an on-the-fly manner with the assumption of non-parallel execution. This section gives a brief overview of BE-PUM architecture and its implementation in Java.

2.2.1 BE-PUM Architecture

The architure of BE-PUM is illustrated in Figure 2.3[2]. Overall, BE-PUM consists of three main components: symbolic execution, binary emulation and CFG storage. The abstraction of x86 execution environment state is expressed by tuple $\langle Register, Flag, Memory, Stack \rangle$. BE-PUM incorporates JakStab 0.8.3 as its binary disassembler and Z3 4.3 as theorem prover. At each time step, a symbolic state at the end of an explored execution path is analyzed by Single-Step Symbolic Execution to determine the next possible execution step. If the instruction on the current state is a data instruction whose destination address is statically decided, BE-PUM simply dissembles the next instruction. If the instruction is a control instruction, BE-PUM performs concolic testing to figure out the next location address. After each exploration step, a new CFG node or a new CFG edge will be created and is stored in CFG storage. The emulation of the x86 instruction set therefore plays a crucial role in state transitioning.

The binary emulation in BE-PUM is shown in Figure 2.4. Each state transition includes a pre-condition (Path Condition P), a post-condition (Path Condition P') and the execution of the instruction itself which follows a sequence of instruction fetching, instruction decoding, operand fetching, then executing and storing the results.

2.2.2 BE-PUM Code Base

Overview

The implementation of BE-PUM is written in Java and compilable with OpenJDKv1.8 [3]. As BE-PUM includes source code of Jakstab 0.8.3 as its disassembler, the total number of Java code files in BE-PUM code project is 2950, containing 2538 classes. Among which we focus on the module that implements the instruction set. Currently, BE-PUM supports 200 x86 instructions [19] with 120 instructions whose semantics are emulated in separate Java code files. The instructions that share similar operations are grouped *manually* into one Java file. As such, we primarily focus on trying fully-automatic code generation competing against those 120 separate files.

Since the instruction set emulation is a component of a larger code project, from software development perspective, it is crucial to understand the structure of the project including the organization of code files and modules in order to continue writing correct codes. A typical static modeling method called Class Diagram [20] is often used to describe structure of a system in terms of programmable object-oriented classes, attributes, operations and relationships among the system's entities. Hence, reconstructing class diagram from an existing code base is one of the ways to overview its structure. The class diagram excerpt in Figure 2.5 shows the implementation choice for an instruction in BE-PUM. We present instruction `AAA` and `CLC` as an example.

[2]These figures are redrawn from paper [3].

[3]https://openjdk.org/projects/jdk8/

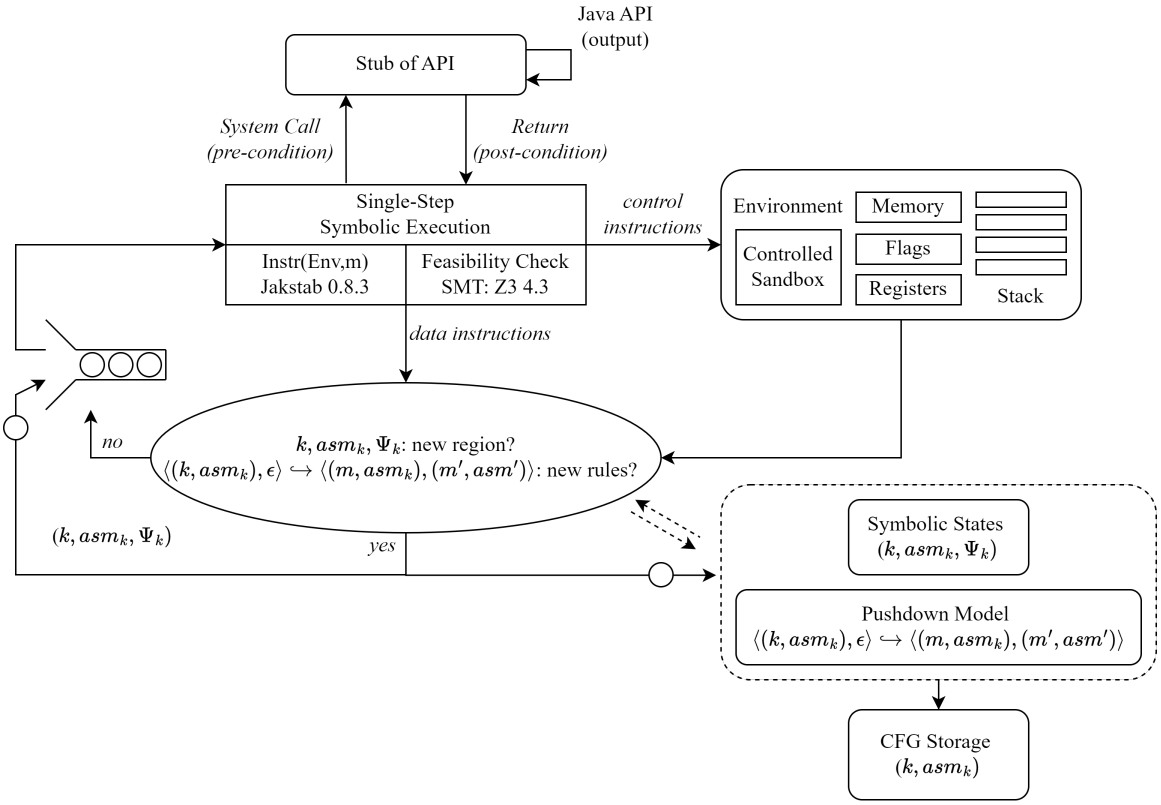


Figure 2.3: BE-PUM architecture [3].

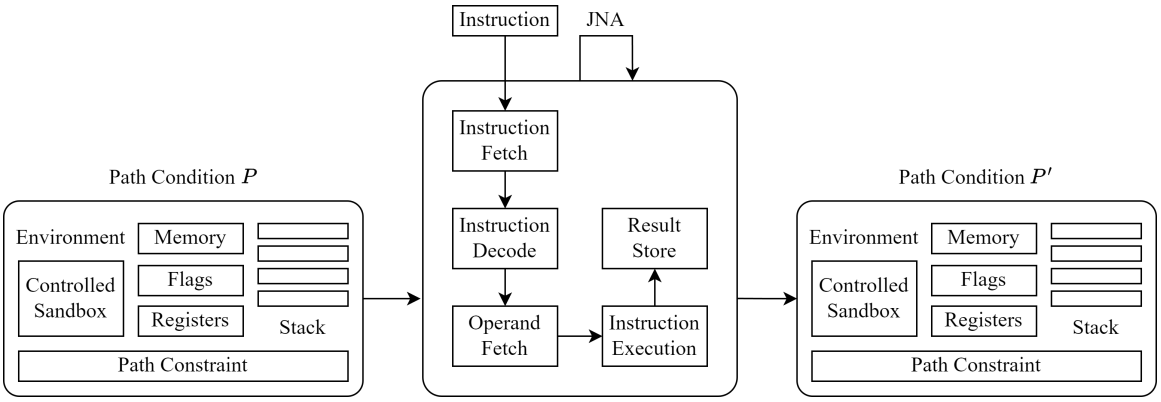


Figure 2.4: Binary emulation in BE-PUM [3].

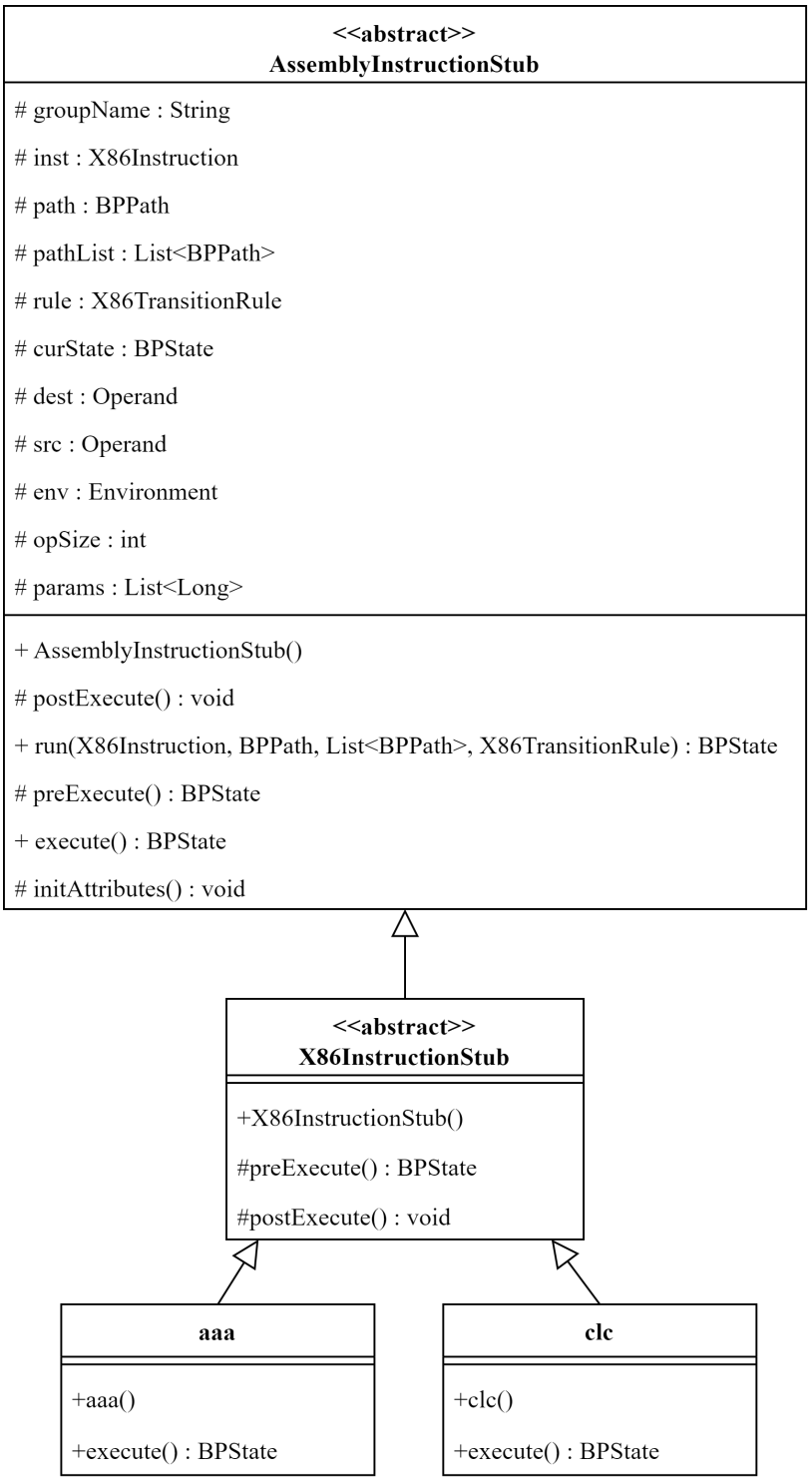


Figure 2.5: Excerpt of class diagram describing components involved in the implementation of x86 instruction.

In the given class diagram segment, the two classes `aaa` and `clc` are designed as concrete implementations which extend the abstract class `X86InstructionStub`. The `execute()` method, which returns a `BPState`, is where the semantics of each instruction is described. The design allows for polymorphic behavior where instances of `aaa` and `clc` can be treated as X86InstructionStub types. With respect to the binary emulation provided in Figure 2.4, organizing the implementation of each instruction with polymorphism makes the call to targeted instruction more dynamic in the sense that the input parameters and input operands need not to be explicitly specified. Instead, they are available internally by the parent class and are ready for access at execution time. This conveniently enables the program to decide at runtime which instruction to invoke. The system, therefore, is easier to extend for the implementation of more instructions. This also serves as our reference point for constructing the Java template that the Code Writer in our system will follow.

Project-level Context of BE-PUM

Like context of a sentence, context of a code project consists of the existing codes and its organization [13]. The representations of the existing codes can be code lines, code fragments or the entire code files. Meanwhile, the organization of the code base can be presented in form of UML diagrams [20]. Since software design and development practices has high degree of modularity and code reuse, incremental implementation to an on-going code project often makes use of existing modules, functions, libraries, or components to avoid duplicating code, saving time and resources. Therefore, in our case, it is necessary that the newly generated code of our system should adhere to the current BE-PUM project-level context, which is to correctly use resources provided by other classes such as variables and methods.

One of the ways to let the generative components know the current project-level context of BE-PUM is to put the raw code base or the project's class diagram into the context of their prompts. However, local LLMs such as ours have limited context length, thus putting content of 2950 Java code files in BE-PUM or class diagram of 2538 classes in one prompt is not feasible. Therefore, given the said code base of BE-PUM, we choose to utilize the project-level context of BE-PUM at the level of individual code lines. The existing code lines in BE-PUM are from these two categories:

1. The implemented code written by BE-PUM's developers.
2. The code snippets explored by traversing BE-PUM's class diagram, which may not yet be written down by the developers.

Chapter 3

Language Model for Text Generation

3.1 Language Modeling

In tasks that involve sequence generation such as Automatic Speech Recognition and Machine Translation, the chosen generated sentence is desired to be more linguistically correct than the others. To achieve it, one of the solutions is to judge whether a generated sequence of tokens is more common than its counterparts within the scope of a particular corpus. Note that these tokens can also be characters, set of characters (also called sub-words), byte-level representation, etc. To quantify a sequence's quality of being probable, one can use statistical approaches where an occurrence of a sequence is considered as a statistical event and thus can be assigned a probability. The computing entity that can assign probability to a sequence with respect to a corpus is known as a language model [21].

In linguistics, sentence is defined as the highest unit in the constituent hierarchy that expresses the minimal syntactic relations between the words it contains [22], or to say: sentence of words is the realization of a language's grammatical syntax and thus induces linguistic context among the words. As an approximation, we can just quantify the relation between the target word and the words that precede it. Therefore, the probability model of a sentence can be expressed by a multiplication chain of conditional probabilities:

$$P(w_1, w_2, \ldots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)..P(w_n|w_1, \ldots, w_{n-1}) \tag{3.1}$$

Sentences are varying in length and order-wise, and have complex context relations among pairs of words. Therefore, to compute the exact value of the said joint probability is intractable in general case. A popular solution to this problem is to approximate it using a type of stochastic model known as discrete Markov chain [23] where the context needed to compute the probability for a target word w_n is limited to $N-1$ words $w_{n-N+1}..w_{n-1}$ preceding it [24]:

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-N+1:n-1}) \qquad \text{for } n \geq N > 0 \tag{3.2}$$

And thus the joint probability for the sequence of words is:

$$P(w_{1:n}) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}) \tag{3.3}$$

The family of this solution is called N-gram language model in which N is usually chosen to be 1, 2, 3, 4 and 5 (uni-grams, bi-grams, tri-grams, 4-grams and 5-grams, respectively). Since probability value ranges from 0 to 1 inclusively, sum of log probabilities is used to avoid vanishing products.

To compute the probability, Maximum Likelihood Estimation (MLE) method counts the occurrence of an n-gram text over the entire corpus and normalized it by the total number of occurrences of $n-1$ words that precede the target word. The training result of the MLE model is a mapping between the unigrams up to n-gram texts and their frequency in the corpus. As this method relies on counting consecutive groups of words, the model MLE would falsely assign zero probability for new sequences that are not presented in the training corpus even if they are linguistically correct. To overcome the issue, smoothing methods have been introduced to shift some of the mass probability to the unseen words.

Nonetheless, probability model (3.1) laid the foundation for the task of modeling language. Starting from the probability model computed merely by occurrences of tokens such as N-grams, the development of language model moved on to the incorporation of handcrafted word features such as part-of-speech and usage frequency, which can be seen in sequential models like vanilla RNNs [25], LSTM [26], and GRUs [27]. Facing the challenge of transmitting long-range dependency, the attention mechanism was introduced to efficiently capture relationship between tokens by looking at all input tokens at once instead of sequential processing. It then became a key concept for the next generation of language models which is the Transformers. Some notable models are BERT(-base) with 110 million parameters (110M) [28] and GPT (117M) [29]. Since then, advances in pre-training and fine-tuning techniques continued to push the boundaries of language models. Nowadays, large-scale transformer-based language models like GPT-3 (175B) and open-sourced Llama [30] dominate the field, setting new benchmarks in language understanding, retrieval and generation. More recently, LLMs such as GPT-4 and Gemini are made capable of understanding multi-modal information.

3.1.1 Masked Language Model

Usually for written text, the meaning of a word does not necessarily depend only on its preceding words [31]. Take an example of a cloze test as follows: "She _____ noodles yesterday." Based on the context given by words that come after the blank, we can predict the missing word to be the action of consuming a type of food (*noodles*) and the verb should be in past tense (*yesterday*). One suitable word to fill in the blank would be *ate*. Based on this philosophy, Devlin et al. [28] proposed Bidirectional Encoder Representations from Transformers (BERT) - a novel deep bidirectional representation of language by conditioning on the context from both sides of a word. As the name suggests, the building block of BERT is a multi-layer Transformer encoder whose core component is based on Attention Mechanism that is called Multi-Head

Attention [32]. Rather than employing recurrence scheme as Recurrent Neural Network (RNN) (recurring left-to-right or bidirection in a sequential order), Transformer model relies solely on just Attention Mechanism to learn the mapping between the input and output sequences, which greatly helps parallelize computation and combat long-range dependency.

There are two stages in applying BERT to a Natural Language Processing (NLP) task: pre-training and fine-tuning. Usually, the first stage is done universally on large unlabeled corpus, which involves training its Masked Language Model (MLM) to learn an embedding of tokens. The training of BERT's MLM is motivated by the cloze test problem where roughly 15% input tokens are randomly masked out and prediction for the masked position is penalized by cross-entropy loss. While RNN learns word order through step-by-step recurrence, BERT relies on its Positional Embedding layer to attain such information. Therefore, a BERT model is usually coupled with a special tokenizer that tokenizes and translates text input into its positional index in the embedding layer and vice versa. The learnt embedding can be used for many down-stream tasks with labeled data such as text classification or Part-of-Speech tagging.

Our study makes use of CodeBERT [33] - a bimodal pre-trained model for both programming languages and natural languages, built on the BERT architecture. It is trained on a large corpus of code from GitHub repositories and has been widely adopted for various code-related tasks such as code generation, code summarization, and code search. In this work, we employ CodeBERT for code search.

3.1.2 Causal Language Model

Unlike masked language models, causal language models predict the next token in a sequence based solely on the preceding tokens. This means that the model generates text one token at a time, which uses the previously generated tokens as context. This sequential process is known as *autoregressive*, which is why causal language models are often referred to as *autoregressive models*. Their training objective is to minimize the difference between the predicted and actual next tokens. The attention mechanism is also used in masked language models. However, instead of attending to all tokens in the input sequence at once, it only focuses on tokens that come before the position being predicted.

The first causal language model in the literature is GPT. GPT utilizes a multi-layer transformer decoder, a variant of the transformer architecture, with an attention mechanism that only spans a windowed context of tokens. GPT also undergoes unsupervised pre-training where unlabeled text corpus is used to optimize next-word prediction objective. During training supervised downstream tasks such as text similarity, question answering, and commonsense reasoning, the auto-regressive language modeling objective is incorporated alongside the regular objective function of the labeled discriminative task. For inference in generative tasks, the sequence of tokens generated by the model is fed back as input to generate the next token. The generation halts when the model generate an end-of-sentence symbol or reaches a pre-defined number of tokens. Based on the success of GPT, GPT-2 [34] explores the capability of GPT models for zero-shot generation task. A zero-shot generation task involves generating text or solving a problem without having been explicitly trained on that specific task or provided with labeled examples during training. Instead, the model

relies on its pre-trained knowledge. Certain cues and requirements are provided, so-called *prompts* to guide the generation. This then lays foundation for GPT-3 [35] - a larger scale of GPT architecture in terms of number of parameters and pre-train data volume.

In our study, we choose to use CodeLlama [36], a causal language model that employs transformer decoder architecture but includes several significant enhancements in its internal operations and pre-trained specifically for code-related tasks such as code completion, translation, summarization, and debugging. The main reason we select CodeLlama is that CodeLlama, or models using CodeLlama as their backbone, is enlisted as state-of-the-art generative models for coding tasks [6] while being offered at reasonable size, with the smallest having 7 billion parameters. Additionally, constraints of time and computing resources also influences our choice. Nevertheless, the generative model component in our work can be replaced with other code-specific models if applicable.

3.2 The Task of Text Generation

Natural language processing includes two primary tasks: natural language understanding and natural language generation. Natural language generation focuses on systems that produce plausible language output. Nowadays, natural language generation extends to broader targets including programming languages, musical notation, mathematical notation and protein sequence [37]. Hence, we refer to the task of language generation as text generation thereafter. Some typical example use cases of text generation include machine translation, digital assistant (chatbot), code generation, visual description and creative story writing.

3.2.1 Text Generation Basics

In [38], text generation task is formulated as follows: Let $\mathcal{P}$ denote the set of desired properties of certain text generation task such as grammatical correctness, semantic accuracy, fluency, language formality, etc. The task of text generation is to obtain a finite sequence of tokens $\boldsymbol{y} = \langle y_1, y_2, \ldots, y_m \rangle$ given a finite input sequence $\boldsymbol{x} = \langle x_1, x_2, \ldots, y_n \rangle$ following the formula:

$$\boldsymbol{y} = f_{\mathcal{M}}(\boldsymbol{x}, \mathcal{P}) \tag{3.4}$$

where $f_{\mathcal{M}}$ is the generative model.

Depending on the type of input data $\boldsymbol{x}$ and the property set $\mathcal{P}$, text generation covers various use cases including:

- When $\boldsymbol{x}$ is not provided or a random-valued vector, the task becomes causal language modelling.
- When $\boldsymbol{x}$ is structured data such as tables, graphs, and databases, the task becomes data-to-text generation.
- When $\boldsymbol{x}$ is of different modality than text, one of the apparent use cases is the captioning task for multi-modality input.

- When $\boldsymbol{x}$ is sequence of text, which is the most common scenario, the task encompasses multiple use case such as translation, summarization, (dialog) question and answering system, and code generation.

The length of $\boldsymbol{x}$ is often called context length - the number of input tokens a language model can handle in one pass, e.g: the initial release of CodeLlama-2 allows a context length of 4096 tokens [36]. There are methods that increase the original context length such as Positional Interpolation which increases it to 32K tokens [39], it however, requires more memory usage in training .

The operation of a typical text generative model can be formulated as follows:

- At training time: The objective function $J(\boldsymbol{\theta})$ is to maximize the probability in predicting next token $\hat{y}_t$ given the previously predicted sequence of token $\boldsymbol{y}_{<t}$. The probability is computed using an embedding model parameterized by $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = -\sum_{t=1}^{T} \log P(\hat{y}_t | \hat{\boldsymbol{y}}_{<t}, \boldsymbol{\theta}) \tag{3.5}$$

- At inference time: a decoding algorithm $g(\cdot)$ is needed to select a token from the learned the distribution $P(Y_t | \boldsymbol{Y}_{<\boldsymbol{t}}, \hat{\boldsymbol{\theta}})$:

$$\hat{y}_t = g(P(Y_t | \boldsymbol{Y}_{<\boldsymbol{t}}, \hat{\boldsymbol{\theta}})) \tag{3.6}$$

Text generation can be further divided into subcategories based on the open-endedness quality - that is the diversity of the output space of the problem. The spectrum of the open-endedness ranges from restricted output space - non-open-ended or closed-ended generation (e.g: text translation, summarization) to liberated output space - open-ended generation (e.g: story generation). Our task, project-level text-to-code generation, is placed at medium open-endedness as the generated code must conform to both code context of the given project and a specification, while the content is still allowed to have certain degree of freedom such as naming new variables and methods, as well as organizing code into smaller, manageable, and reusable components.

Closed-ended and open-ended problems require different architectures. For closed-ended tasks, the most common approach is the encoder-decoder model, where the auto-regressive model serves as the decoder and the encoder is usually a bidirectional language embedding model. For open-ended tasks, a decoder-only model is often used. The boundary of using these two architectures is often not clear. For example, an encoder-only model [40] or decoder-only model [35] is proved to perform on par or even better than encoder-decoder models for language translation. Likewise, encoder-decoder models can be used for open-ended tasks but due to achieving the similar performance to a decoder-only architecture, this approach is not resource-efficient and hence the latter approach is more favorable. One of the major problems related to the property of open-endedness is called Constrained Generation or Constrained Decoding - that is how well the generative language model adheres to the response format requirements specified in the prompts while still maintained the semantics of its answers. This is an on-going issue in the literature, addressed in recent study [41, 42]. Specifically, for the problem of code generation such as ours, several formats must be upheld such as grammar rules of the target programming language, syntax of file-level or project-level code context.

3.2.2 Decoding Procedure in Text Generation

As mentioned earlier, the conditional probability learned from autoregressive models requires a decoding algorithm to generate tokens. There have been multiple designs for the decoding algorithm. Starting with the most straightforward approach - greedy sampling - research on decoding schemes has led to improvements. Techniques such as beam search, sampling methods, and advanced algorithms like nucleus sampling and top-k sampling have been developed to enhance the quality and diversity of generated text. These methods address the limitations of greedy selection by exploring a broader range of possible outputs and optimizing the trade-off between accuracy and creativity.

Greedy Sampling

In greedy selection, during each step of the text generation process, the model picks the token with the highest predicted probability. This approach is simple and quick, but the resulted output sequences are often suboptimal as the method falls short for choosing tokens with locally high probabilities while neglecting the total probability of the sequence.

Beam-search Sampling

While decoding, beam-search sampling employs greedy selection but instead of storing one output sequence, beam-search keeps a number of output sequences by keeping k predicted candidates at each time step.

These two ways of decoding relying on maximum probability is feasible for closed-ended task. For open-ended tasks, it creates repetitive generation problem regardless of model scale [43]. To reduce repetition:

- N-gram blocking [44]: N-gram blocking is the most simple method, to avoid repeating N-grams in the generation sequence.
- Coverage loss [45]: The method modifies training objective such that it prevents the attention mechanism from attending to the same words over again.

Probabilistic Sampling

However, to make the generated text sequence more natural to human, always choosing the tokens with high probabilities restrict the creativity and diversity of the speaker [43]. Hence, probabilistic sampling methods are used. Sampling is the process of selecting a subset of individuals from a larger population to make observations and draw inferences about that population, often the selection is tied with a probability distribution. There are several typical sampling methods used in recent literature:

- Vanilla sampling: All tokens in the vocabulary have a chance to be selected even the probability is small.
- Top-k sampling: Only a subset of tokens in the vocabulary whose probabilities are in the top-k largest have a chance to be selected. Increasing k gives diverse output, while decreasing k provides more consistent output.
- Top-p sampling: Only a subset of tokens in the vocabulary whose probabilities, when ranked in descending order, sum up to a probability mass of p. As top-k

sampling requires defining a fixed k beforehand, which is not suitable for very skewed or flat distributions, top-p sampling, in fact, makes k dynamic.

- More complex sampling techniques: Typical sampling [46] - re-weights the score based on the entropy of the distribution; Epsilon sampling [47] - sets a threshold for lower bound valid probabilities.

3.3 Efficient Fine-tuning Techniques for Large Language Models

While embodied general knowledge from massive pre-trained corpus, (large) language models often need to be further fine-tuned to serve more specialized domains. With such a huge number of parameters, consumer GPUs may not be able to accommodate the conventional fine-tuning process. For example, the model CodeLlama-2-7B-Instruct has 7 billion parameters. At precision float32, each parameter value takes 32 bits, which accounts for a total of approximately 28GB for the whole model. Meanwhile, regular consumer GPU memory often comes with equal to or less than 24GB. Therefore, to facilitate study on large language models for the mass, it is essential to develop memory-efficient training techniques. A direct method to reduce memory consumption when loading large model is called n-bit (or b-bit) quantization in which floating-point precision is quantified into b-bit representation [48]. On the other hand, technique used in reducing memory footprint during training is called Parameter-Efficient Fine-Tuning (PEFT) [49, 50, 8]. PEFT involves adjusting only a subset of model parameters rather than the entire model while maintaining performance.

3.3.1 Quantization

The backbone of Neural networks is matrix-matrix multiplication, which is further broken down into parallel matrix-vector multiplications by neural network accelerators, for example, GPUs. There are two basic components in one matrix-vector multiplication: 1) The processing elements $C_{n,m}$ and 2) The accumulators A_n. Figure 3.1 demonstrate an example of multiplying weight matrix $\boldsymbol{\theta} \in \mathbb{R}^{2\times 3}$ and a row vector $\boldsymbol{x} \in \mathbb{R}^3$ using processing elements C and accumulators A.

The computation starts with loading the bias term b_n into the accumulator A_n. Then weight $\boldsymbol{\theta}_{n,m}$ and input vector x_m are loaded into the corresponding arrays and their product is computed by $C_{n,m} = \boldsymbol{\theta}_{n,m} x_m$ in one cycle. The value at the accumulator A_n is then computed by

$$A_n = b_n + \sum_m C_{n,m} \tag{3.7}$$

The operation described above is known as Multiply-Accumulate (MAC). This step is repeatedly performed for larger matrix-vector multiplications. After all cycles are completed, the values in the accumulators are transferred back to memory to be utilized in the subsequent neural network layer. Using precision float32 would place a heavy load in bit transfer as well as causing high energy consumption for MAC operation, especially for large neural network like large language models.

The purpose of model quantization is to convert floating point values into fixed-point precision using a lower number of bits, which is then empirically proved to consume

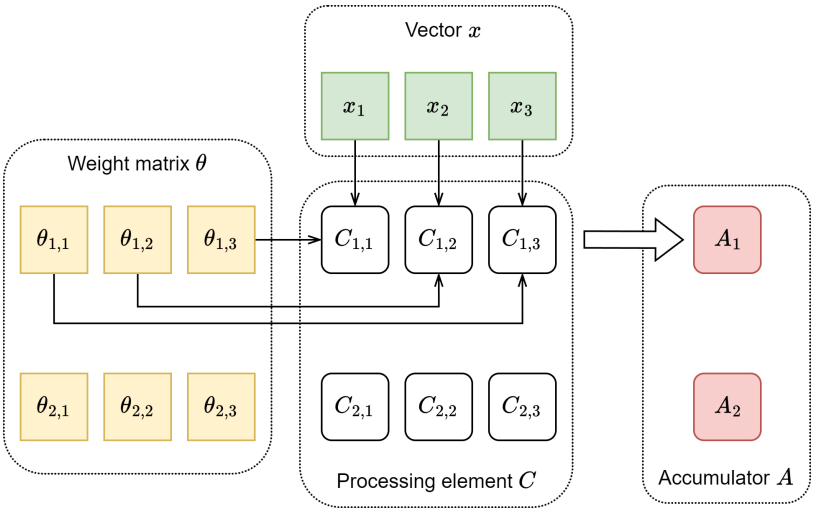


Figure 3.1: Matrix-vector multiplication of neural network accelerators [48].

less computation energy and reduce the amount of data transfer [48]. The mechanism of quantization is as follows:

Hypothetically, the original floating-point vector $\boldsymbol{x}$ is approximated by a floating-point scalar $s_{\boldsymbol{x}}$ (so-called scale factor) multiplied by a vector of integers: $\boldsymbol{x} \approx \hat{\boldsymbol{x}} = s_{\boldsymbol{x}} \cdot \boldsymbol{x}_{\text{int}}$. We use different scale factors for weight matrix $\boldsymbol{\theta}$ and vector input $\boldsymbol{x}$, hence Equation 3.7 becomes:

$$\begin{aligned}
\hat{A}_n &= \hat{b}_n + \sum_m \hat{C}_{n,m} \\
&= \hat{b}_n + \sum_m \hat{\boldsymbol{\theta}}_{n,m} \hat{\boldsymbol{x}}_m \\
&= \hat{b}_n + \sum_m (s_{\boldsymbol{w}} \boldsymbol{\theta}^{\text{int}}_{n,m})(s_{\boldsymbol{x}} \boldsymbol{x}^{\text{int}}_m) \\
&= \hat{b}_n + s_{\boldsymbol{w}} s_{\boldsymbol{x}} \sum_m \boldsymbol{\theta}^{\text{int}}_{n,m} \boldsymbol{x}^{\text{int}}_m
\end{aligned} \tag{3.8}$$

To reduce the accumulated errors and avoid overflow, the accumulators are still stored in high bit-width, such as 32-bit. Hence, the bias term $\hat{b}_n$, although being quantized based on scale factors of weight matrix and input vector, is stored in 32-bit integer form. The accumulator value at 32-bit integer is then quantized for the second time into low-bit integer to transfer into the memory. Compared to the raw floating-point MAC (Equation 3.7), quantized MAC (Equation 3.8) has its processing element C computed in low-bit fixed-point precision (e.g: 8-bit, 4-bit) and only at the accumulator, high-bit computation is performed.

Uniform Affine Quantization, or Asymmetric Quantization is the most commonly used quantization technique. The three hyper-parameters needed for the technique are: bit-width b, scale factor s and zero-point z where: b is the number of bit of the target representation, s and z are used for mapping floating-point values into b-bit integers with z used to avoid error of quantizing real zero. The operation is carried out as follows:

Denote $[\cdot]$ as rounding to the nearest integer, the unsigned integer projection of

floating-point value is computed by

$$\boldsymbol{x}^{\text{int}} = \text{clamp}(\left\lceil \frac{\boldsymbol{x}}{s} \right\rceil + z; 0, 2^b - 1) \tag{3.9}$$

where:

$$\text{clamp}(x; l, h) = \begin{cases} l, & x < l \\ x, & l \leq x \leq h \\ h, & x > h \end{cases} \tag{3.10}$$

The original floating-point value is then approximated by

$$\begin{aligned} \boldsymbol{x} \approx \hat{\boldsymbol{x}} &= s(\boldsymbol{x}^{\text{int}} - z) \\ &= s\left[\text{clamp}(\left\lceil \frac{\boldsymbol{x}}{s} \right\rceil + z; 0, 2^b - 1) - z\right] \end{aligned} \tag{3.11}$$

The approximation of $\hat{\boldsymbol{x}}$ is effectively the quantization function of Uniform Affine Quantization, denoted as $q(\boldsymbol{x}; b, s, z)$. We can see that $q_{\min} = -sz$ and $q_{\max} = s(2^b - 1 - z)$. Any value that is higher than these boundaries is then clamped into the nearest boundary. Hence, quantization suffers from the trade-off between *clipping error* and *rounding error*. Consequently, there have been multiple improvements to quantization such as Symmetric Uniform Quantization and Power-of-two quantization [51].

3.3.2 Low-rank Adaptation

While quantization helps reduce memory consumption per network parameter, Low-Rank Adaptation (LoRA) [8] is a PEFT method that reduces the number of trainable parameters in fine-tuning by freezing weights of the pre-trained model and injects trainable rank-decomposition weight matrices into each layer of it.

The nature of fine-tuning is that given the original weight matrix $\boldsymbol{\theta}_0 \in R^{d \times k}$ of the pre-trained model, the new weight after fune-tuning is in fact $\boldsymbol{\theta}_0 + \Delta\boldsymbol{\theta}$. Let matrices $B \in R^{d \times r}$ and $A \in R^{r \times k}$ be the rank-decomposition matrices of $\Delta\boldsymbol{\theta}$ with rank r of choice, $r \ll min\,(d, k)$, we have:

$$\Delta\boldsymbol{\theta} = BA \tag{3.12}$$

During forward pass, both $\Delta\boldsymbol{\theta}$ and $\Delta\boldsymbol{\theta} = BA$ are multiplied with the same input x and the two outputs are summed element-wise:

$$h = \boldsymbol{\theta}_0 x + \Delta\boldsymbol{\theta} x = \boldsymbol{\theta}_0 x + BAx \tag{3.13}$$

The visualization of this decomposition is shown in Figure 3.2 [1].

Before attempting retraining, the value of $\Delta\boldsymbol{\theta}$ should be zero, hence, one of the two matrices is initialized with zeros. A common practice is to initialize the other with random values from gaussian distribution or other pre-trained LoRA weights such as PiSSA [52] and LoftQ [53]. Fine-tuning can lead to issues of underfitting or overfitting, which are considered undesirable deviations from the pre-trained model.

[1] This figure is adapted from paper [8]

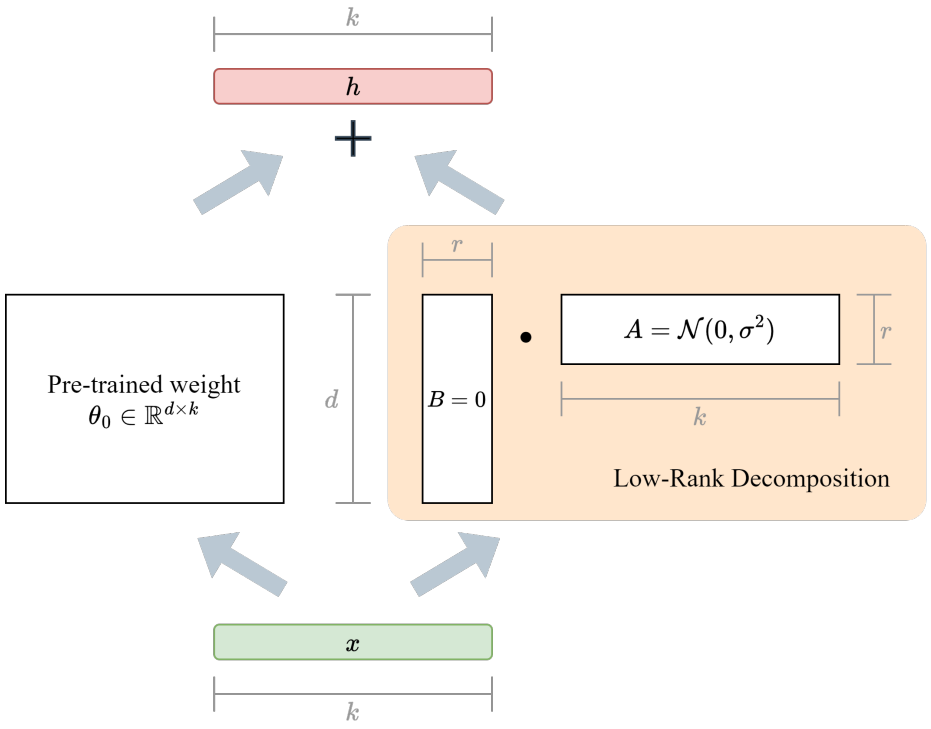


Figure 3.2: Injection of Low-Rank Adaptation into pre-trained weight.

A common solution is to reduce the magnitude of $\Delta\boldsymbol{\theta}$ by applying a hyperparameter scaling factor. Equation 3.13 hence becomes:

$$h = \boldsymbol{\theta}_0 x + \frac{\alpha}{r}\Delta\boldsymbol{\theta}x = \boldsymbol{\theta}_0 x + \frac{\alpha}{r}BAx \tag{3.14}$$

where alpha is typically chosen to be a multiple of r. During backpropagation, since only $\Delta\boldsymbol{\theta}$ receives gradient updates, the optimizer consumes much less GPU memory to store states of trainable parameters.

Empirical study shows that for model fine-tuning, adjusting only a subset of parameters is sufficient to achieve performance that is comparable to a fully fine-tuned model [54]. Consequently, it is unnecessary to inject LoRA modules into every parameter layer of the pre-trained model; rather, they can be applied to a selected subset of layers. The determination of which subset of parameter layers to consider is often done by empirical experiments. According to the findings presented in [8], it has become standard practice to inject LoRA modules into the linear layers of the attention mechanism in large language models such as the four query, key, value and output head layer ($\boldsymbol{\theta}_q$, $\boldsymbol{\theta}_k$, $\boldsymbol{\theta}_v$, $\boldsymbol{\theta}_o$).

3.3.3 Technical Usage of Efficient Fine-tuning Techniques

Regarding technical implementation, the injection of LoRA adapter, as well as quantization operation, is supported via machine learning platform and libraries, such as PyTorch. Figure 3.3 briefly visualizes the hierarchy of related machine learning tools as well as presents a common scenario of machine learning research using such tools. We start with PyTorch [2] and TensorFlow [3] being the foundational open-source

[2]https://pytorch.org
[3]https://www.tensorflow.org/

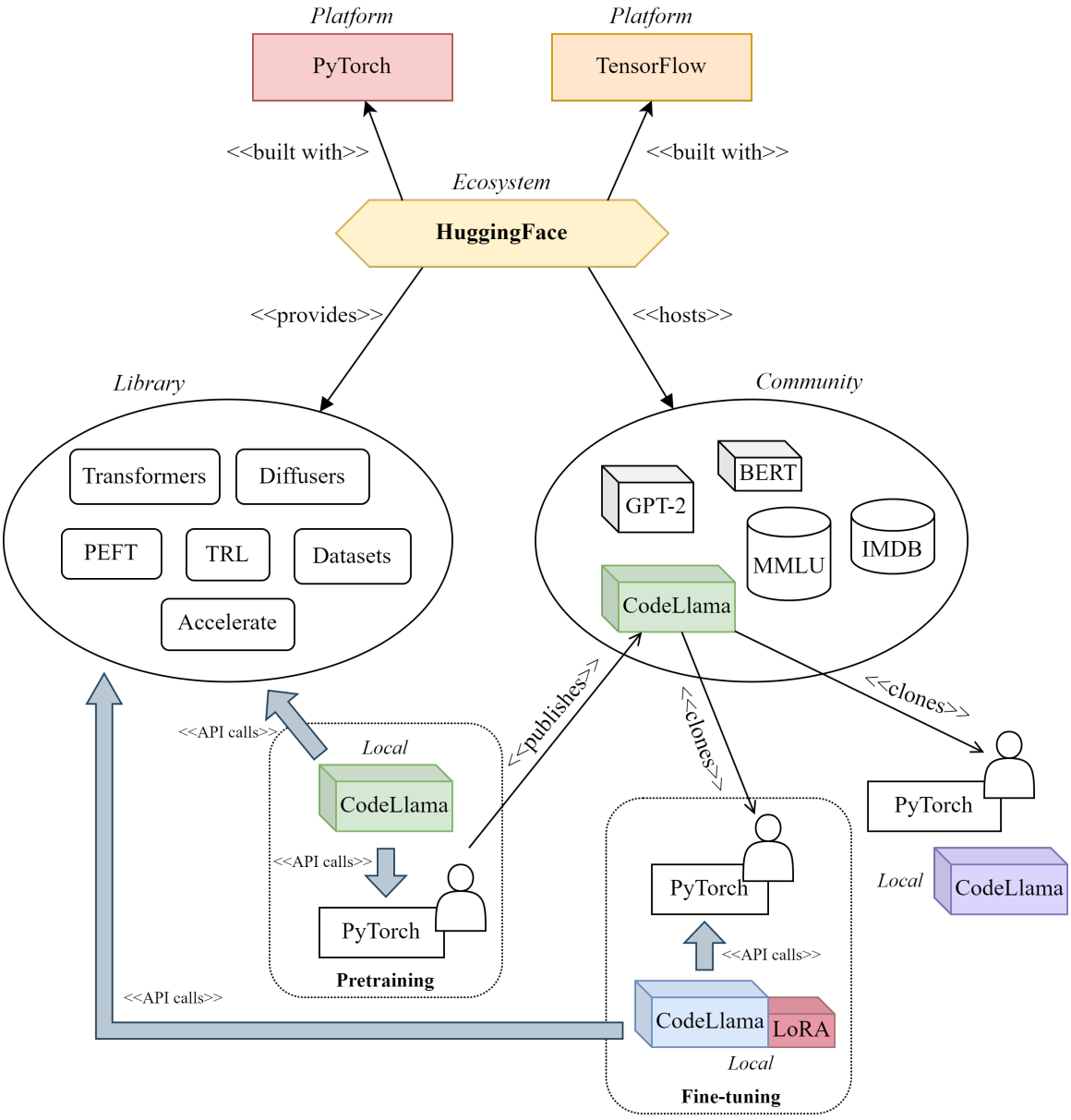


Figure 3.3: Machine learning research with HuggingFace ecosystem.

machine learning platforms that provide users with APIs such as tensor operations, GPU acceleration, neural network training and deployment. HuggingFace, on the other hand, is not a platform but rather an ecosystem that leverages existing platforms to provide more specialized libraries such as those for Transformers, Diffusers, etc. Besides offering machine learning community its open-source libraries, HuggingFace also hosts a large number of shared machine learning artifacts such as trained models and datasets. Figure 3.3 shows that for pretraining case - that is one can use both specialized APIs from HuggingFace's libraries as well as native PyTorch's APIs to construct and train a model from scratch. The trained model can then be publish to the community hub for open-source sharing. Our case - that is to fine-tune a model, for example CodeLlama, clones the shared model to our local machine. Then by using APIs provided by the ecosystem of HuggingFace and PyTorch, we can add in our modifications to the cloned model's architecture, e.g: inserting LoRA adapter into a subset of layers of CodeLlama and integrating the model into TRL - a Transformer Reinforcement Learning pipeline to fine-tune it.

The benefits of employing the discussed efficient fine-tuning techniques includes:

- Drastic reduction in memory and storage usage. For instance, our study employs model CodeLlama-2-7B-Instruct which originally comes with 28GB at full precision. After applying quantization and LoRA injection, the total model sizes at only 4.8GB, accounting for only 17.14% memory usage of the pre-trained model's.
- Multi-tasking by switching between different LoRA layers on the same frozen pre-trained model. This approach is theoretically feasible; however, the current version of the machine learning tools in use only supports it during inference, not during training.

While these methods offer significant benefits and have become standard practices in machine learning research with large models, they also have drawbacks. Quantization can introduce approximation issues such as clipping errors and rounding errors, whereas fine-tuning with LoRA adaptation may experience a margin of latency.

Chapter 4

Reinforcement Learning with Proximal Policy Optimization

4.1 Reinforcement Learning Basics

4.1.1 Reinforcement Learning in Machine Learning Hierarchy

Depending on the types of training data available to the learning models, the specified learning objectives and types of feedback, machine learning can be divided into three major paradigms: supervised learning, unsupervised learning and reinforcement learning [55, 56].

- Supervised learning:
  - Data: The given data is fully annotated by external supervisors. Each data point includes an input and its known ground-truth output.
  - Objective: The model is expected to learn a mapping from inputs to outputs that can be used to predict labels for new and unseen inputs of the same knowledge domain.
  - Feedback: Direct feedback is provided in the form of labels for each input. Often, it is represented as a loss function measuring the error distance between ground-truth value and model's output.
  - Examples: classification, regression, and object detection.
- Unsupervised learning:
  - Data: The given data has no label.
  - Objective: The model is expected to discover hidden patterns or structures within the data, such as clusters, associations, or underlying distributions.
  - Feedback: There is no direct feedback in terms of correct answers. Instead, based on the inherent structure of the data, some proxy measurements are used such as inter-cluster and intra-cluster distances.
  - Example: Clustering, data mining, abnormality detection.

- Reinforcement learning:
  - Data: The training data has no label and is generated through interactions between the agent and the environment. Often, no test data or testing phase is needed.
  - Objective: The model is expected to learn a decision strategy (formally called policy - a mapping from states to actions) that maximizes cumulative reward over time.
  - Feedback: The feedback is indirect and often delayed through rewards or penalties based on actions taken over time.
  - Examples: Game playing like Go and Chess, robotic arm control, self-driving cars.

Among the three, reinforcement learning is favored for problems where the relationship between actions and outcomes is not smooth or continuous. Apart from the aforementioned examples, another renowned application that inspires our study is the creation of InstructGPT [57] in which reinforcement learning from human feedback is used to effectively adjust large language model's response . For the instance of our work, instead of human feedback, we use feedback from software verification tool like the compiler to assess the generated code's correctness.

4.1.2 Elements of Reinforcement Learning

Reinforcement learning is a framework for solving decision-making problems. A typical reinforcement learning problem is represented in the form of an agent learning to make optimal decisions within an environment by interacting with it and improving through feedback in the form of cumulative rewards [55, 56]. It is specified by 6 core elements: Value function, Policy, Reward, (Environment) model, Exploration – exploitation balancing, and Representation [55]. We document 4 major elements below:

Environment

The environment that an agent acts within defines 4 following components:

- State space $\mathcal{S}$: The set of all environment configurations of interest. $\mathcal{S}$ can be discrete or continuous.
- Action space $\mathcal{A}$: The set of all possible actions the agent could perform within the environment. $\mathcal{A}$ can also be discrete or continuous. The construction of action space is also affected by the nature of the agent, e.g: what actions the agent can perform.
- Environment model:
  - State transition probability $\mathcal{P}(s_{t+1}|s_t, a_t)$: The probability of moving from one state to another when taking a specific action.
  - Reward function $\mathcal{R}(a, s) \in \mathbb{R}$: A function that provides feedback from the environment to the agent. It maps a state or state-action pair to a numerical reward r. A reward is a signal that immediately informs the agent whether its chosen action is good or bad at each time step.

Based on the visibility of the environment to the agent, an environment can be classified as either:

- Perfect information: The agent can observe a complete view of the environment states and the components of the environment are deterministic. E.g: the chess board in chess playing, the map in traveling saleman problem, the maze in maze solving problem.
- Imperfect information: The environment contains stochastic elements, hidden information and/or the agent cannot observe all states of the environment at once but rather it is generated as the agent explores. E.g: poker, stock trading, self-driving cars' surroundings.

Policy

Policy $\pi(\cdot)$ is the decision strategy used by the agent to determine the next action to take given the current state, so-called the mapping from states to actions. A policy can be deterministic $\pi(s) = a$ or probabilistic $\pi(a|s) = P(a|s)$. As taking action transitions the agent to the next state, which obtains a reward, policy optimization is one of the methods that aim to find an optimal mapping from states to actions.

Value Function

While the reward function provides immediate feedback after taking an action, the value function estimates the total future reward that the agent can expect to receive starting from the next state.

The future reward g_t, so-called the return, is the sum of the discounted rewards obtained by moving to the terminal state by k more steps, starting from the next state s_{t+1}:

$$g_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{k-1} r_{t+k} = \sum_{i=0}^{k-1} \gamma^i r_{t+1+i} \tag{4.1}$$

where $\gamma \in (0, 1]$ is the discount factor that weights down the importance of the future rewards because they may have higher uncertainty and not provide immediate benefits.

The state-value function $v_\pi(s_t)$ of state s_t is then given by the expected return g_t as $v_\pi(s_t) = E[g_t|s_t]$. Often, it is denoted using generalized notation as follows:

$$v_\pi(s) = \mathbb{E}[g_t|S_t = s] \tag{4.2}$$

Additionally, another type of value function that involves predicting the return of a state-action pair is called state-action-value function and is defined by:

$$q_\pi(s, a) = \mathbb{E}[g_t|S_t = s, A_t = a] \tag{4.3}$$

State-value function differs from state-action-value function in the sense that state-value function estimates the return if the agent transitions to the next states at $t+1$ onwards following the guidance of current policy $\pi(a|s)$. Meanwhile, state-action-value function measures the expected return if the agent actively chooses to move to some arbitrary state in the next time step $t+1$ and then back to following the guidance of the current policy $\pi(a|s)$ for time step $t+2$ onwards. State-action-value function helps in exploration as it tries taking new actions.

Reward

The reward is a component of the environment model as aforementioned and is often a scalar. The point of time that the reward is given to the agent is crucial, as it influences how the agent learns to improve its actions. One significant challenge in this context is known as sparse reward. A sparse reward in reinforcement learning refers to a situation where an agent receives feedback or rewards infrequently or only under specific conditions. This can make the learning process more divergent as the agent has to explore a large number of actions or states before discovering those that lead to rewards. Characteristics of sparse rewards include: 1) Infrequent rewards - that is rewards are given only occasionally, rather than after every action or state transition; and 2) Delayed feedback - that is a significant delay between taking an action and receiving a reward, which makes it hard for the agent to learn which actions are beneficial.

Text Generation as a Reinforcement Learning Problem

When framing text generation as a reinforcement learning problem, the generative model acts as a policy. Hence the learnt probability distribution over output token represents the decision strategy that serves for later decoding process. There is no physical environment, instead, the state space is considered to include all the generated string obtained after taking a decoding step. The reward is straightforward - that is the scalar value resulted from assessing the generated output sequence.

The generative model with decoder architecture like CodeLlama, when fine-tuning with reinforcement learning, is capped with a special layer called a Value Head. Policy optimization like PPO would depend on calculating the advantage of taking a specific action (selecting a token) in a given state. This calculation involves subtracting the value of being in the state from the value of the state-action pair. The additional value-head layer projects the final hidden states onto a scalar to estimate the state's value.

4.1.3 An Example

Figure 4.1 demonstrates relationship between state-value function and state-action-value function in an example setting. Note that the superscript $(^{(1,1),\ (1,2),\ldots})$ is used for branch indexing, e.g: $s_{t+1}^{(1,2)}$ denotes the state at time step $t+1$ resulting from taking action branch #1 and is a child state at index #2.

The example scenarios is as follows: Supposed that the agent exploration has reached state s_t and received reward r_t. In this state, there are 2 possible actions that the agent can take: $a_t^{(1)}$ and $a_t^{(2)}$ whose probabilities are decided by the stochastic policy $\pi(a|s)$. If $a_t^{(1)}$ is taken next, there are chances that it might end up in state $s_{t+1}^{(1,1)}$, $s_{t+1}^{(1,2)}$ or $s_{t+1}^{(1,3)}$ and would receive some future reward $r_{t+1}^{(1,i)}$ accordingly $(i = 1, 2, 3)$. On the other hand, if $a_t^{(2)}$ is taken next, the agent could end up in state $s_{t+1}^{(2,1)}$ and receive reward $r_{t+1}^{(2,1)}$. As this is a random process, to assess which move is more probable to give higher benefit in the long run, at the current state s_t we calculate $v_\pi(s_t)$ - the weighted average (the expected) value of the future rewards of all possible exploration paths starting from s_t.

Applying Equation 4.2 for this instance, we have:

$$v_\pi(s_t) = \sum_i \pi(a_t^{(i)}|s_t) q_\pi(s_t, a_t^{(i)}) \tag{4.4}$$

The gray box representing the state-action value $q_\pi(s, a)$. Applying Equation 4.3, we have:

$$\begin{aligned} q_\pi(s_t, a_t^{(i)}) &= \sum_j p_t^{(i,j)} (r_{t+1}^{(i,j)} + v_\pi(s_{t+1}^{(i,j)})) \\ &= \sum_j p(s_{t+1}^{(i,j)}, r_{t+1}^{(i,j)} | s_t, a_t^{(i)}) (r_{t+1}^{(i,j)} + v_\pi(s_{t+1}^{(i,j)})) \end{aligned} \tag{4.5}$$

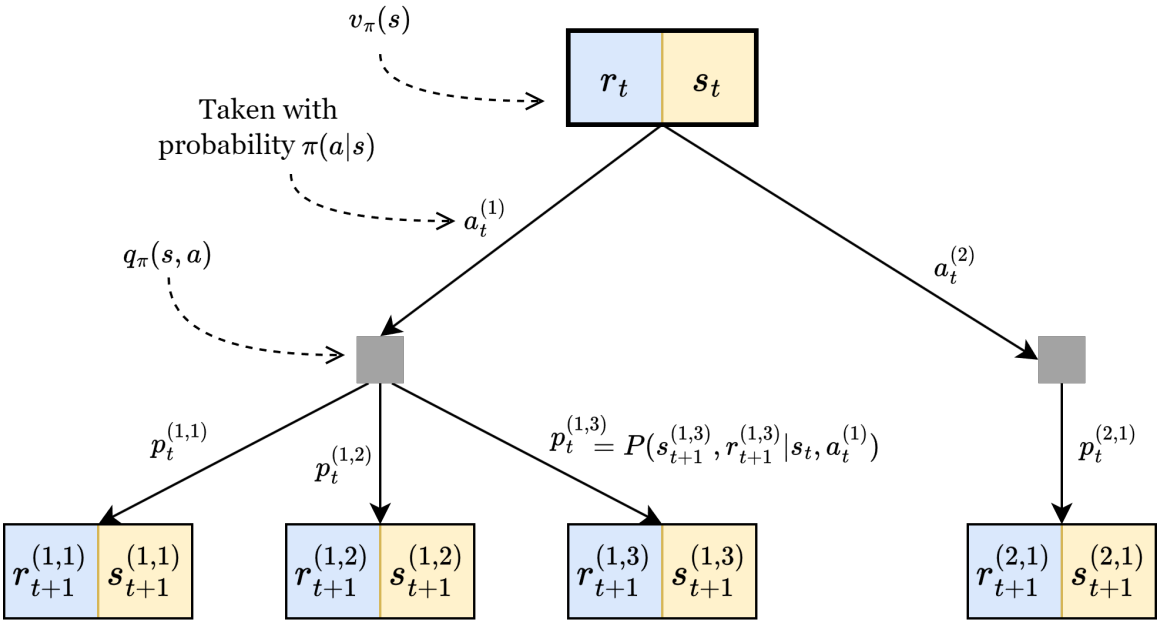


Figure 4.1: An example of state-action-reward exploration.

Finally, at state s_t, to see if taking action $a_t^{(1)}$ yields higher future reward than the average action suggested by policy $\pi(a|s)$, we simply take the difference between $q_\pi(s_t, a_t^{(1)})$ and $v_\pi(s_t)$. This quantity is called "Advantage function".

Often, state-value and action-state-value functions are written without the expectation notation. Hence, the generalized formulas are:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \quad \text{(from Equation 4.4)} \tag{4.6}$$

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a)(r + v_\pi(s')) \quad \text{(from Equation 4.5)} \tag{4.7}$$

Putting it all together in a dynamic view, we have: reinforcement learning involves an agent, at each time step t, is given an environment state $s_t \in \mathcal{S}$ and decides to perform action $a_t \in \mathcal{A}$ with respect to its learned behavior $\pi(\cdot)$. The agent then receives a reward r_t according to $\mathcal{R}(a, s)$ and is presented with a new state s_{t+1}

according to $\mathcal{P}(s_{t+1}|s_t, a_t)$. A sequence of interactions over time $t = 1, 2, \ldots, T$ is called a trajectory (or a trial) that ends in terminal state S_T:

$$s_1, a_1, r_2, s_2, a_2, r_3, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$$

A reinforcement learning problem can either be episodic or continuous. In episodic settings, the interaction of the agent reaches a terminal state and restarts to predefined standard initial states. If the transition of states fits the property of a Markov process, the reinforcement learning problem can be formulated as Markov Decision Process (MDP) [55, 58].

4.2 Proximal Policy Optimization

The objective of reinforcement learning boils down to the aim of maximizing the accumulative reward. There are two major lines of methods: state-value-based and policy-based. State-value methods try to optimize $v_\pi(s)$ or $q_\pi(s, a)$ and then use it to guide the selection of actions. Meanwhile, policy-based approaches directly optimize the parameterized policy function $\pi_{\boldsymbol{\theta}}(a|s)$ for $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ which is then used to select actions without consulting a value function. A value function is only used to learn the policy parameters. Compared to value-based methods, policy-based methods with parameterization is claimed to yield a superior results and allow injecting prior knowledge [58]. Proximal Policy Optimization is a type of policy-based methods and has become the most commonly used reinforcement learning optimization method [55]. As such, our work employs Proximal Policy Optimization to optimize generative models using reinforcement reward signal.

The introduction to Proximal Policy Optimization would be extensive as it is an improvement built upon a series of algorithms developed over the history of policy-based reinforcement learning. Nonetheless, the foundation of policy-based methods is Policy Gradient.

Let $J(\boldsymbol{\theta}) \triangleq v_{\pi_{\boldsymbol{\theta}}}(s)$ be a scalar function that measures the accumulative reward obtained by following $\pi_\theta(a|s)$ starting from an initial state s. The objective of reinforcement learning is expressed as the maximization of $J(\boldsymbol{\theta})$. The update of θ is carried out by gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \frac{\partial J(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t} \tag{4.8}$$

The Policy Gradient Theorem for episodic cases states that:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \propto \sum_s \mu_\pi(s) \sum_a q_\pi(s, a, \boldsymbol{\theta}) \frac{\partial \pi(a|s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \tag{4.9}$$

The proof for this theorem is present in Appendix A.

The standard policy gradient as presented in Equation A.5 is unbiased. This means on average, the estimated gradient used to update the policy is equal to the true gradient and that the gradient accurately estimates the direction in which the policy should be updated to maximize the expected reward. However, these gradient estimates have high variance - the estimates can vary significantly from one update to the next,

leading to instability and inefficiency in the learning process. Consequently, there have been numerous improvements proposed to overcome the issue such as Asynchronous Advantage Actor-Critic (A3C) [59], Trust Region Policy Optimization (TRPO) [60], PPO [61] and Phasic Policy Gradient (PPG) [62]. Among which PPO is a direct enhanced version of TRPO.

The current literature replaces the raw state-action-value function with advantage function, hence Equation A.5 is often referred to as:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}}}, a\sim\pi_{\boldsymbol{\theta}}} \left[\hat{A}(s, a, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \ln \pi(a|s, \boldsymbol{\theta}) \right] \tag{4.10}$$

In TRPO, a ratio of new and old policy is used in place of the policy term, which further modifies the original objective function $J(\boldsymbol{\theta})$ to the following:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}}}, a\sim\pi_{\boldsymbol{\theta}}} \left[\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) \frac{\pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta}_{\text{old}})} \right] \tag{4.11}$$

The gradient update is then scaled with ratio between new and old policies instead of the raw new policy. However, the high variance problem still remains if the new policy has much larger value. To constraint the distance between two probability distributions $\pi_{\text{old}}(\cdot|s)$ and $\pi(\cdot|s)$ to a δ amount, Kullback–Leibler (KL) divergence is used:

$$\mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\text{KL}(\pi_{\text{old}}(\cdot|s), \pi(\cdot|s)) \right] \leq \delta \tag{4.12}$$

PPO - KL Penalty Version

Empirically, TRPO performs worse than standard policy gradient method on certain problems with unclear reasons and complex to implement [61]. To overcome the issue, PPO is introduced. At first, it is a slight adjustment to TRPO in the way that it still keeps the KL divergence term but instead incorporates it into the objective function as a penalty weighted by hyperparamter β:

$$J_{\text{KL}}(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}}}, a\sim\pi_{\boldsymbol{\theta}}} \left[\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) \frac{\pi(a|s, \boldsymbol{\theta})}{\pi(a|s, \boldsymbol{\theta}_{\text{old}})} \right] - \beta \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}_{\text{old}}}}} \left[\text{KL}(\pi_{\text{old}}(\cdot|s), \pi(\cdot|s)) \right] \tag{4.13}$$

PPO - Clipping Version

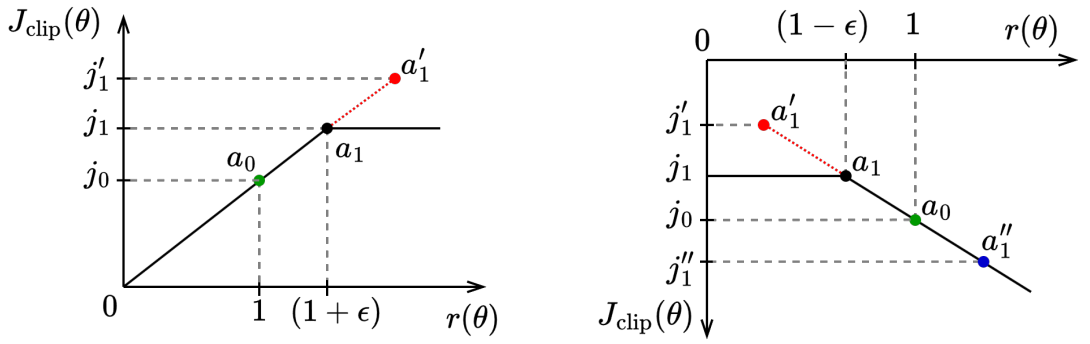
Firstly, let us denote $r(\boldsymbol{\theta}) = \frac{\pi(a|s,\boldsymbol{\theta})}{\pi(a|s,\boldsymbol{\theta}_{\text{old}})}$. So $r(\boldsymbol{\theta}_{\text{old}}) = 1$. Equation 4.11 becomes:

$$J(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}}}, a\sim\pi_{\boldsymbol{\theta}}} \left[\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) r(\boldsymbol{\theta}) \right] \tag{4.14}$$

Instead of relying on the value of ratio $r(\boldsymbol{\theta})$, the Clipping version of PPO clips it within a range of $[1 - \epsilon, 1 + \epsilon]$. However, to retain the true value of $r(\boldsymbol{\theta})$ when it is sufficiently small, the objective function is framed as follows:

$$J_{\text{clip}}(\boldsymbol{\theta}) = \mathbb{E}_{s\sim\mu_{\pi_{\boldsymbol{\theta}}}, a\sim\pi_{\boldsymbol{\theta}}} \left[\min \left[r(\boldsymbol{\theta}) \hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}),\ \text{clip}(r(\boldsymbol{\theta}), 1 - \epsilon, 1 + \epsilon) \hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) \right] \right] \tag{4.15}$$

The effect of using clipping on the objective function $J_{\text{clip}}(\boldsymbol{\theta})$ is demonstrated in Figure 4.2. Supposed that the initialization with action a_0 causes $r(\boldsymbol{\theta}_{\text{old}}) = 1$.



(a) $\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) \geq 0$ (b) $\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}}) < 0$

Figure 4.2: Clipping effect on $J_{\text{clip}}(\boldsymbol{\theta})$.

- Figure 4.2a presents the case when the estimated advantage reward $\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}})$ is positive. This indicates that the action $a'_1 \sim \pi'(a|s, \boldsymbol{\theta})$ the agent has just performed is a good action. However, it also shows that $\pi'(a|s, \boldsymbol{\theta})$ is greater than $\pi(a|s, \boldsymbol{\theta}_{\text{old}})$. In the case where $r(\boldsymbol{\theta})$ grows very large, gradient $\nabla J(\boldsymbol{\theta}) = j'_1 - j_0$ also grows significantly. Applying it to update the policy's parameters using Equation 4.8 would change the old policy drastically, which might then make the objective function overshoot the convergence point. To combat this situation, PPO clips $r(\boldsymbol{\theta})$ to a value of $1 + \epsilon$. By this way $\nabla J(\boldsymbol{\theta}) = j_1 - j_0$ is kept small enough such that the parameters changing steadily towards good actions.
- Figure 4.2b illustrates the case when $\hat{A}(s, a, \boldsymbol{\theta}_{\text{old}})$ is negative. This means that the agent has just performed a bad action, supposedly $a'_1 \sim \pi'(a|s, \boldsymbol{\theta})$. As $\nabla J(\boldsymbol{\theta}) = j'_1 - j_0$ is a positive value, by the policy's parameters update Equation 4.8, we actually encourage the bad action a'_1 to happen again in the future since the gradient has now ascended. To reduce this effect, PPO clips $r(\boldsymbol{\theta})$ to a small value of $1 - \epsilon$, which reduces the gap between j'_1 and j_0 (to just $j_1 - j_0$). This hypothetically makes bad action a'_1 less likely to happen compared to when there is no clipping. Using $\min(\cdot)$ operator without a lower bound, PPO lets bad action $a''_1 \sim \pi''(a|s, \boldsymbol{\theta})$ happen. This turns out not an issue since for this case the gradient is negative. By the policy's parameters update Equation 4.8, negative gradients allow the objective function to move back towards the convergence point. Even if it overshoots the convergence point, it continues to be adjusted by gradient ascent.

The Clipping version of PPO is more widely used than its KL Penalty counterpart. The pseudocode for the Clipping version is given in Algorithm 1.

PPO method does face drawbacks when dealing with sparse rewards. These issues are addressed by alternating between KL Penalty version and Clipping version, which are proposed by Hsu et al. [63]. One important characteristic of PPO despite sparse rewards is that it hypothetically allows arbitrary scalar value of rewards as the objective function uses ratio between new and old policies. However, in practice, it is preferable to scale the scalar rewards to a certain range.

Algorithm 1 PPO - Clipping Version

procedure POLICY_UPDATE($\mathcal{S}, \mathcal{A}, \mathcal{R}, \pi(a|s,\boldsymbol{\theta}), N, T, M$)
  $\pi(a|s,\boldsymbol{\theta}_{\text{old}}) \leftarrow \pi(a|s,\boldsymbol{\theta})$
  for i $= 1 \ldots$ N **do**
    Run $\pi(a|s,\boldsymbol{\theta})$ for T time steps on $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$.
    Compute $\hat{A}(s,a,\boldsymbol{\theta}_{\texttt{old}})$ for all time steps.
    for j $= 1 \ldots$ M **do** ▷ SGD: Stochastic Gradient Descent
      Do SGD on $-J_{\texttt{clip}}(\boldsymbol{\theta})$ with $\pi(a|s,\boldsymbol{\theta}_{\texttt{old}})$ and $\pi(a|s,\boldsymbol{\theta})$.
      Update $\boldsymbol{\theta}$ accordingly.
    end for
    $\pi(a|s,\boldsymbol{\theta}_{\text{old}}) \leftarrow \pi(a|s,\boldsymbol{\theta})$
  end for
  return $\pi(a|s,\boldsymbol{\theta})$
end procedure

Chapter 5

Implementation

5.1 CoDeb System Overview

Our proposed system for project-level text-to-code generation via reinforcement learning on compiler feedback is called *CoDeb* consisting of 4 major components: Code Writer, Code Debugger, Reward Function and BEPUM-KB. Figure 5.1 gives an overview of the system.

Taking inspiration from *Pair programming* - a collaborative programming practice where two developers work together on the same code, our system consists of two generative language models where one takes on the roles of code writing (Code Writer) and the other is responsible for code debugging (Code Debugger). The two undergo reinforcement learning with a Reward Function using PPO method. The knowledge base BEPUM-KB containing BE-PUM code snippets assists Code Debugger in its solution finding process. This section documents the details of each component along with their input and output.

5.2 Description on Input and Output

5.2.1 Input

As can be seen from Figure 5.1, the input into the system consists of two information:

- Complete description of an x86 instruction in natural language which includes *Description* section and *Flag update* section of an x86 instruction's specification (as stated in Section 2.1.2). It also includes short description of each instruction's variant if applicable.
- A prepared Java code template of the target class that once filled in by the system, should emulate the specified instruction.

In our case, we have extracted the class diagram of BE-PUM with the help of IntelliJ IDEA [1] - an Integrated Development Environment (IDE) for Java. An excerpt of BE-PUM class diagram is shown in Figure 2.5. From class diagram, the generation of the corresponding Java class can be done automatically using a simple diagram parser. As such, the Java template input into the system relies on the design choice of

[1] https://www.jetbrains.com

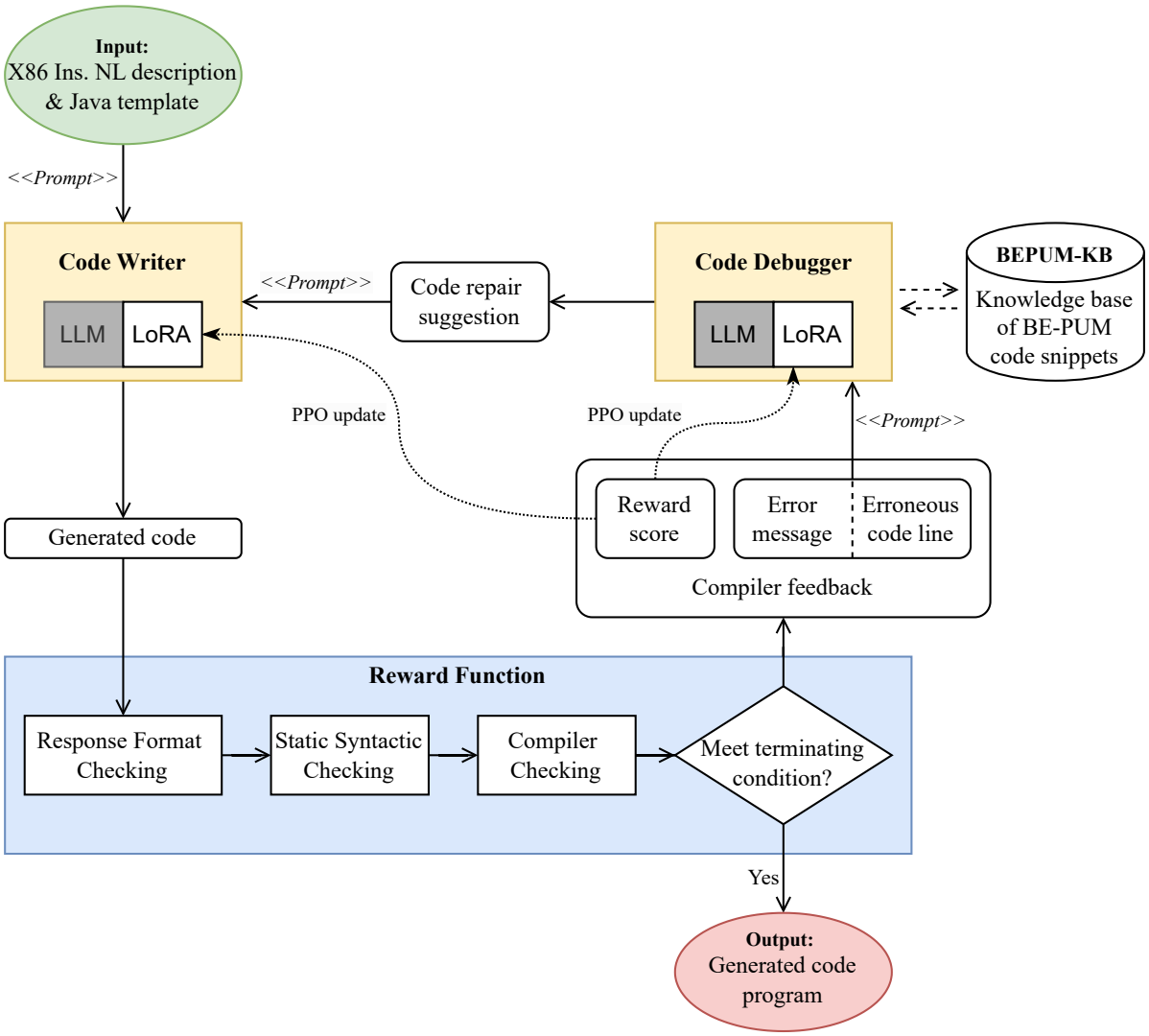


Figure 5.1: CoDeb system for project-level text-to-code generation.

the emulation project and can be automatically obtained. We reserve one Java class for each x86 instruction and consider its text description as software specification for the class. In addition to the generated Java template, we also pre-retrieve available variables resulting from code inheritance and necessary environment variables for an instruction to execute on.

5.2.2 Output

The expected output of CoDeb system is a complete Java code file that emulates the requested x86 instruction. More specifically, the output Java code file should contain the prepared content along with the generated codes filled in correct position. An example output is shown in Figure B.1.

5.3 Code Writer

5.3.1 Model Construction

Base model. Following the typical parameter-efficient fine-tuning process, the component Code Writer consists of a frozen LLM injected with a LoRA adapter. In our implementation, we choose *CodeLlama-7b-Instruct-hf* [2] - a variant of CodeLlama specialized in instruction-following tasks, with 7 billion parameters in size. However, the LLM slot can be substituted with any code LLM of choice that is compatible with PEFT fine-tuning. The generation configuration settings are as follows: sampling is enabled, with a temperature of 1. The minimum number of new tokens is set to 80% of the minimum length, while the maximum number of new tokens is set to 250% of the minimum length. Additionally, the top-k value is set to 50, the top-p value is 0.95, and the number of return sequences is 1. The minimum length is the total number of tokens of the given Java code template, which is 566 tokens.

Configuration for LoRA adapter. The LoRA adapter is configured for the type of causal language modeling with rank $r = 8$, $\alpha = 8$ and one of its low-rank matrices' weight initialized with values drawn from gaussian distribution. The target layers of the frozen LLM injected with LoRA are components of attention layers, specifically the linear key, query, and value layers, namely `q_proj, k_proj, v_proj` and an output projection layer `o_proj`.

Configuration for quantization. The model is quantized with 4-bit quantization scheme, resulting in 4.8GB model size for the Code Writer.

Configuration for reinforcement fine-tuning with PPO. The PPO configuration is set as follows: the learning rate is fixed at $1.41 \times 10^{-}5$, with a batch size and mini-batch size both set to 1. Gradient accumulation steps are also set to 1, while the number of PPO epochs is 4. The target KL divergence is 0.5, and score scaling is enabled.

5.3.2 Response Format

As the heavy lifting of implementing a PEFT-LLM is done by PyTorch framework, the crucial part that is use-case specific is how to construct the input prompt and

[2] https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf

govern the format of the response regarding syntax and semantics. The reason we choose Instruct version of CodeLlama is to enable more systematic request for code generation as well as to help constrain the output format since the Code Writer should first produce static-syntactically correct Java code file.

Since the entirety of the Code Writer's response is expected to be valid code content of a Java code file, we define the response format of the Code Writer as follows:

```
    ```java
    <Java code content>
    ```
```

The response format includes only one pair of tokens ```` ```java ```` and ```` ``` ````, along with non-null `<Java code content>`. The reason that the special tokens denoting the code section are needed because generally, the generative models are trained to further explain and/or provide summarization of its coding answers. For small local model like *CodeLlama-7b-Instruct-hf*, specifying the exclusion of such text explanation from its answer does not always guarantee the desired outcome. As only the Java programmable code content is needed, we mark it with special tokens in order to easily separate it out from the text explanation, and later serve the penalization.

This response format then decides the construction of the prompt as well as the Reward Function used for reinforcement fine-tuning the Code Writer.

5.3.3 Prompt Construction

The standard prompt format used for conversational instruction version of CodeLlama 2 [3] is given as:

```
<s>[INST] <<SYS>>
{{ system_prompt }}
<</SYS>>

{{ u1 }} [/INST] {{ m1 }} </s><s>[INST] {{ u2 }} [/INST]
```

Note that the content of each round of conversation including one request from user (`u1`) and one response from the model (`m1`) is wrapped within the two tags `<s> </s>`. The part of the prompt that starts with `<s>` without ending in `</s>` is then followed by the generation of the model. Usually it is the last round of conversation in the prompt.

Here we only use one conversation round for each time of prompting with the following prompt structure:

```
<s>[INST] <<SYS>>
{{ system_prompt }}
<</SYS>>

{{ u1 }} [/INST]
```

There are two scenarios where the Code Writer is asked to generate code:

- Initial code generation: The Code Writer receives the system's input described in Section 5.2.1 and is asked to generate a static-syntactically parsable Java code file.

[3] Article on CodeLlama-2 prompting: https://huggingface.co/blog/llama2#how-to-prompt-llama-2

- Iterative code generation: The Code Writer receives an erroneous code file and the suggestion from the Code Debugger on how to fix such error. It is then asked to correct the error, the output must still be a static-syntactically parsable Java code file.

Initial Code Generation

For our first generation round, the content of `{{ system_prompt }}` is manually provided as follows:

```
You are an expert Java programmer.
Complete the code given in the context.
Your answer should contain only Java codes with no textual explanation.
Your answer should start with ‘‘‘java mark and end with ‘‘‘ mark.
Provide comment for each block of codes.
```

The content of user's first request `{{ u1 }}`:

```
Your task is to implement x86 instruction named {{inst_name}}.

### Input:
{{text_description}}

### Context:
Generate codes for function {{func}} in this Java code:
{{additional_knowledge}}

### Response:
```

The value for `{{inst_name}}` is the uppercase sequence of an instruction's name.

The value for `{{text_description}}` is the first information of the system's input (described in Section 5.2.1) that is the description of each x86 instruction in natural language, obtained from parsing the ISA manual of x86. The text includes a general description regarding operation, flag changes, and short description of each instruction's variant. The paragraph structure of the raw text is retained as is.

The second information of the system's input (described in Section 5.2.1) is used as below:

- The value for `{{func}}` is the name of the Java method that the Code Writer should try to complete.
- The value for `{{additional_knowledge}}` making the context for in-context learning is the code template of the target Java class. The declaration of package name, import statements and class structure is automatically retrieved from the class diagram of the project. We manually declare several code statements to demonstrate syntax for input retrieval. A sample is shown below:

```
package v2.org.analysis.transition_rule.x86instruction;

import v2.org.analysis.environment.memory.MemoryV2;
import v2.org.analysis.environment.stack.StackV2;
import v2.org.analysis.path.BPState;
import v2.org.analysis.transition_rule.stub.X86InstructionStub;
import java.util.List;
import org.jakstab.asm.Operand;
import org.jakstab.asm.x86.X86Instruction;
import v2.org.analysis.environment.Environment;
import v2.org.analysis.path.BPPath;
import v2.org.analysis.transition_rule.X86TransitionRule;
import v2.org.analysis.value.LongValue;
```

```
import v2.org.analysis.value.BooleanValue;

public class <classname> extends X86InstructionStub {
    @Override
    public BPState execute() {
        // From Top-most base class
        String groupName = this.groupName;
        X86Instruction inst = this.inst;
        BPPath path = this.path;
        List<BPPath> pathList = this.pathList;
        X86TransitionRule rule = this.rule;
        BPState curState = this.curState;
        Operand dest = this.dest;
        Operand src = this.src;
        Environment env = this.env;
        int opSize = this.opSize;
        List<Long> params = this.params;

        // Example syntax for retrieving register values, use them for other registers.
        LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
        LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");

        // Example syntax for retrieving flag values, use them for other flags.
        BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
        BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();

        // Retrieve stack value
        StackV2 stack = (StackV2) env.getStack();

        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // Generate from here


        return null;
    }
}
```

Iterative Code Generation

For our iterative generation rounds, the content of `{{ system_prompt }}` is manually provided as follows:

```
You are an expert Java programmer.
Your job is to fix an erroneous program given a guidance.
The program that needs correction and the guidance are in the input.
Use suggestion in the context.
Output the corrected version of the given program only.
Do not explain anything before or after the program.
Your answer should start with ‘‘‘java mark and end with ‘‘‘ mark.
Provide comment for each block of codes.
```

The content of `{{ u1 }}`:

```
### Input:
- The erroneous program <class-name>.java:
<error-code>

- The guidance to fix the program:
<guidance>

### Context:
Suggest using these snippets:
<suggest-snippets>

### Response:
```

where:

- `<class-name>`: The name of the class within the Java code file.
- `<error-code>`: All of the code within the Java code file.
- `<guidance>`: The guidance of the Code Debugger on explaining the error and suggesting possible ways to fix it.
- `<suggest-snippets>` (optional): The code snippets of BE-PUM that is possibly related to the error, fetched from the knowledge base BEPUM-KB.

5.4 Reward Function

Component Reward Function consist of 3 checking procedures, namely Response Format Checking, Static Syntactic Checking and Compiler Checking. Algorithm 2 gives the details of the Reward Function where:

- r: A scalar reward value.
- p: A scalar penalty value.
- δ: A scalar threshold value which is the allowed ratio between newly generated code lines that are comment lines and the total number of newly generated code lines.
- R: The final reward value after taking penalty.

The Reward Function first starts with `RESPONSE_FORMAT_CHECKING` procedure (Algorithm 3) to ensure that the code section in the response of the Code Writer can be obtained. Note that the `STATIC_SYNTACTIC_CHECKING` procedure (Algorithm 4) calls to `COMPILER_CHECKING` (Algorithm 5) as we only allow sending static-syntactically correct Java codes to the compiler due to two reasons: 1) Static syntactic checking can rely on static parser alone, hence reducing number of calls to and waiting time on running compiler; and 2) The compiler can deliver more concise and meaningful feedback.

Algorithm 2 Reward Function - Scoring the response of Code Writer

procedure REWARD_FUNCTION($\mathbf{s}$)
  $r \leftarrow 0$
  $p \leftarrow 0$
  $\delta \leftarrow 0.7$
  $\langle r, p \rangle \leftarrow$ `RESPONSE_FORMAT_CHECKING`$(\mathbf{s}, r, p)$
  $\langle r, p, \mathbf{e}, \mathbf{s}_{\text{fcom}} \rangle \leftarrow$ `STATIC_SYNTACTIC_CHECKING`$(\mathbf{s}, r, p, \delta)$ ▷ Include Compiler Checking Procedure
  $R \leftarrow r - p$
  return $\langle R, \mathbf{e}, \mathbf{s}_{\text{fcom}} \rangle$
end procedure

5.4.1 Response Format Checking

To guarantee the response format specified in section 5.3.2, as shown in Algorithm 3, this procedure helps determine if the generated text contains actual codes because programmable codes and natural text may be indistinguishable by Java parser like

javalang [4]. Additionally, as LLMs usually try to explain the content of the code which is unwanted, separating out the code section helps pointing out such explanation part and therefore, helps penalizes the behaviour.

Algorithm 3 Checking the format of the response generated by Code Writer

procedure RESPONSE_FORMAT_CHECKING($\mathbf{s}, r, p$)
  $N \leftarrow$ `number of characters in s`
  if **s** begins with ```` ```java ```` **then**
    $r \leftarrow r + 1$
  else
    $p \leftarrow p + 1 + \frac{1}{N}|\mathbf{s}_{0:i\text{```java}}|$ ▷ Number of tokens before reaching the first ```` ```java ````
  end if

  if **s** ends with ```` ``` ```` **then**
    $r \leftarrow r + 1$
  else
    $p \leftarrow p + 1 + \frac{1}{N}|\mathbf{s}_{i\cdots:}|$ ▷ Number of tokens after reaching the last ```` ``` ````
  end if

  if **s** contains only one ```` ```java ```` **then**
    $r \leftarrow r + 1$
  else
    $p \leftarrow p + 1$
  end if

  if **s** contains only two ```` ``` ```` **then**
    $r \leftarrow r + 1$
  else
    $p \leftarrow p + 1$
  end if

  return $\langle r, p \rangle$
end procedure

5.4.2 Static Syntactic Checking

Algorithm 4 details the steps performed in the checking for syntactic correctness of the Code Writer's response. There are two main purposes in this procedure:

- Check for Java static syntactic correctness using a static parser for Java - *javalang*, lines `#4..7,29,30`. First, function `GET_CODE_SECTION` takes out the section of codes from the Code Writer's response based on the token pair ```` ```java ```` and ```` ``` ````. Next, `GET_JAVA_PARSED_CODE` continues to run javalang parser on the extracted code section.
- Examine the generated code content of the Code Writer by multiple criteria:

[4] https://github.com/c2nes/javalang

- Check if the generated code retains all tokens given in the Java template, lines `#8..13`. If all of the tokens in the given template cannot be found in the Code Writer's response, then the total penalty value for this criteria reaches 1.
- Check if the generated code retains the Java template in terms of code lines, lines `#15..17`. This criteria differs from the above in the sense that the above looks for the usage (or reuse) of the template's code tokens such as variable names, method names. Meanwhile, this checks if the manually provided parts which are syntactically correct is unaltered.
- Check if the Code Writer actually generate new code lines, lines `#18..20`. There are cases where the Code Write cheats by only outputting the given template, so we need to penalize this behaviour.
- Check if the all of the newly generated code lines are not comment lines, lines `#21..27`. There are instances where the Code Writer cheats by generating only comment lines. This is similar to the previously mentioned cheat case but also consumes memory space. Hence, penalty is needed for this behaviour.

5.4.3 Compiler Checking

Algorithm 5 shows the steps taken in Compiler verification. First, function `REFINE_CODE` adds missing information to the generated codes. These information includes package name, import statement and correct class name for the target Java code file. Note that these information is fixed and can be automatically obtained from pre-defined coding design, in our case, we take it from the extracted class diagram of the project.

Next, function `COMPILE_IN_BEPUM` inserts the generated Java code file into BE-PUM project in an appropriate directory and calls Java compiler *Javac* to run project-wide compilation. The message $\mathbf{m}$ returns by the compiler Javac is then collected. If the compilation succeeds, an large immediate reward value of 2 is given. Otherwise, we examine the errors and try to correct them.

Inspired from CoTran [64], CoDeb also focuses on fixing the first encountered compilation error at a time. Function `PARSE_ERROR_MESSAGE` parses the message $\mathbf{m}$ and returns the first error's description $\mathbf{e}_f$ and its line number l_f. Using l_f, we try to reward the model with its correctly generated part while penalizing the remaining. Function `GET_NEAREST_COMMENT` retrieves the comment line $\mathbf{s}_{\text{fcom}}$ that is immediately above the erroneous code line (if any). $\mathbf{e}_f$ and $\mathbf{s}_{\text{fcom}}$ are then useful for Code Debugger.

5.5 Code Debugger

5.5.1 Model Construction

The configurations for the base mode, LoRA adapter, quantization and PPO training are similar to those of the Code Writer. The only difference is that the maximum number of new tokens is set at 128 only. Since the response from the Code Debugger is to put into the prompt of the Code Writer, it should be kept short and concise. Ideally, sharing the same frozen LLM with the Code Writer would significantly reduce GPU memory consumption during training compared to running two separate LLMs.

Algorithm 4 Checking the static syntactic parsability of Code Writer's response

procedure STATIC_SYNTACTIC_CHECKING($\mathbf{s}, r, p, \delta$)
  $\mathbf{e}_\text{f} \leftarrow$ NULL ▷ Prepared for Compiler Checking Procedure
  $\mathbf{s}_\text{fcom} \leftarrow$ NULL ▷ Prepared for Compiler Checking Procedure
  $\mathbf{s} \leftarrow$ GET_CODE_SECTION($\mathbf{s}$)
  $\mathbf{s} \leftarrow$ GET_JAVA_PARSED_CODE($\mathbf{s}$)
  if $\mathbf{s}$ is valid **then**
    $r \leftarrow r + 2$

    if $\mathbf{s}$ retains all tokens given in template $\mathbf{s}'$ **then**
      $r \leftarrow r + 1$
    else
      $\mathbf{t}_\text{neg} \leftarrow$ tokens in $\mathbf{s}'$ but not in $\mathbf{s}$
      $p \leftarrow p + \frac{1}{|\text{tokens of } \mathbf{s}'|}|\mathbf{t}_\text{neg}|$
    end if

    $l \leftarrow$ furthest matched line number between $\mathbf{s}$ and $\mathbf{s}'$

    Criteria: Code Writer retains the template in terms of lines
    $r \leftarrow r + \frac{l}{|\text{lines of } \mathbf{s}'|}$
    $p \leftarrow p + 1 - \frac{l}{|\text{lines of } \mathbf{s}'|}$

    Criteria: Code Writer should generate more lines
    $r \leftarrow r + 1 - \frac{l}{|\text{lines of } \mathbf{s}|}$
    $p \leftarrow p + \frac{l}{|\text{lines of } \mathbf{s}|}$

    Criteria: Prevent new generated code lines are all comments
    $c \leftarrow$ number of new lines that are comments
    if $\frac{c}{|\text{lines of } \mathbf{s}| - l} \leq \delta$ **then**
      $r \leftarrow r + 2$
    else
      $p \leftarrow p + 2$
    end if

    $\langle r, p, \mathbf{e}, \mathbf{s}_\text{fcom} \rangle \leftarrow$ COMPILER_CHECKING($\mathbf{s}, r, p$)

  else
    $p \leftarrow p + 2$
  end if

  return $\langle r, p, \mathbf{e}, \mathbf{s}_\text{fcom} \rangle$
end procedure

Algorithm 5 Checking the compilability of Code Writer's response in BE-PUM

procedure COMPILER_CHECKING($\mathbf{s}, r, p$)
  $\mathbf{s} \leftarrow$ REFINE_CODE($\mathbf{s}$)
  $\mathbf{m} \leftarrow$ COMPILE_IN_BEPUM($\mathbf{s}$)
  if $\mathbf{m}$ has status SUCCESS **then**
    $r \leftarrow r + 2$
  else
    $\langle \mathbf{e}_f, l_f \rangle \leftarrow$ PARSE_ERROR_MESSAGE($\mathbf{m}$)
    $r_{\text{correct\_part}} \leftarrow \frac{l_f}{|\text{lines of } \mathbf{s}|}$
    $r \leftarrow r + r_{\text{correct\_part}}$
    $p \leftarrow p + 1 - r_{\text{correct\_part}}$
    $\mathbf{s}_{\text{fcom}} \leftarrow$ GET_NEAREST_COMMENT($l_f, \mathbf{s}$)
  end if
  return $\langle r, p, \mathbf{e}, \mathbf{s}_{\text{fcom}} \rangle$
end procedure

However, the PyTorch platform no longer supports this training scheme with multiple adapters in a single session. Hence, there are two possible ways of implementation:

- Constructing the Code Writer and Code Debugger as two separate frozen LLMs, each equipped with its own LoRA adapter.
- Using one frozen LLM injected with one LoRA adapter for both code writing and debugging tasks (multi-task learner).

There is no restriction on the response format for the Code Debugger, indicating that the required formats for the tasks of writing and debugging differ. Consequently, the initial implementation, which separates the two models, appears intuitively effective. However, given that large language models (LLMs) are capable of multi-task language understanding [65], the alternative implementation is also considered in our experiments.

5.5.2 Prompt Construction

Following the same format used for a single turn of conversation prompting in the Code Writer, the complete prompt template for the Code Debugger is manually prepared as follows:

```
<s>[INST] <<SYS>>
You are an expert Java code debugger.
Your job is to suggest solutions to fix an erroneous program.
The program that needs correction and its error are in the input.
The suggestion is in the context.
Note that class name is changeable while file name cannot be changed.
<</SYS>>

### Input:
- The errorneous program <class-name>.java:
<error-code>

- The error message:
<error-message>

### Context:
Suggest using these snippets:
<suggest-snippets>

```

```
### Response:
[/INST]
```

where:

- <class-name> is substituted with the target instruction's name in lowercase.
- <error-code> is replaced with the entire faulty program generated by the Code Writer.
- <error-message> is substituted with the first encountered error's message $\mathbf{e}_f$ informed by the compiler via the Reward Function.
- <suggest-snippets> is the code snippets in BE-PUM that potentially express the same semantics as the generated code line where the first encountered error occurs. The code line $\mathbf{s}_{\text{fcode}}$ parsed from the error message $\mathbf{e}_f$ and the nearest comment to it $\mathbf{s}_{\text{fcom}}$ together forms a query into BEPUM-KB. The returned top-k most semantically similar code snippets of BE-PUM are then placed in to this placeholder.

An instantiation of the prompt in the case of fixing instruction JLE is given below:

```
<s>[INST] <<SYS>>
You are an expert Java code debugger.
Your job is to suggest solutions to fix an erroneous program.
The program that needs correction and its error are in the input.
The suggestion is in the context.
Note that class name is changeable while file name cannot be changed.
<</SYS>>

### Input:
- The errorneous program jle.java:
package v2.org.analysis.transition_rule.x86instruction;

import v2.org.analysis.environment.memory.MemoryV2;
import v2.org.analysis.environment.stack.StackV2;
import v2.org.analysis.path.BPState;
import v2.org.analysis.transition_rule.stub.X86InstructionStub;
import java.util.List;
import org.jakstab.asm.Operand;
import org.jakstab.asm.x86.X86Instruction;
import v2.org.analysis.environment.Environment;
import v2.org.analysis.path.BPPath;
import v2.org.analysis.transition_rule.X86TransitionRule;
import v2.org.analysis.value.LongValue;
import v2.org.analysis.value.BooleanValue;

public class jle extends X86InstructionStub {
    @Override
    public BPState execute() {
        // From Top-most base class
        String groupName = this.groupName;
        X86Instruction inst = this.inst;
        BPPath path = this.path;
        List<BPPath> pathList = this.pathList;
        X86TransitionRule rule = this.rule;
        BPState curState = this.curState;
        Operand dest = this.dest;
        Operand src = this.src;
        Environment env = this.env;
        int opSize = this.opSize;
        List<Long> params = this.params;

        // Example syntax for retrieving register values, use them for other registers.
        LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
        LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");

        // Example syntax for retrieving flag values, use them for other flags.
        BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
```

```
        BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();

        // Retrieve stack value
        StackV2 stack = (StackV2) env.getStack();

        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        if (CFlag.getValue() == 1 || AFlag.getValue() == 1) {
            // Jump to the destination
            path.setPC(dest.getValue());
            return null;
        } else {
            // Continue executing current instruction
            return null;
        }
    }
}


- The error message:
jle.java:46: error: incomparable types: boolean and int
        if (CFlag.getValue() == 1 || AFlag.getValue() == 1) {
                                  ^


### Context:
Suggest using these snippets:
if (env.getFlag().getPFlag().equal(new BooleanValue(1))) {

### Response:
[/INST]
```

5.6 BEPUM-KB

To inform the Code Writer and Code Debugger with the potentially suitable code syntax from BE-PUM project to use, we need to index the project-level context of BE-PUM for semantic retrieval. The component that holds this information and facilitates semantic searching is named BEPUM-KB (KB: Knowledge Base). This section documents the process of constructing BEPUM-KB, starting from collecting the project-level context of BE-PUM, to embedding and plugging it into a small semantic search engine.

5.6.1 Collection of BE-PUM's Project-level Context

With respect to the two types of project-level context declared in Section 2.2.2, we divide the collection of these code into two scenarios respectively: Collection of existing code base and Collection of explored code lines from class diagram.

Collection of Existing Code Base

All code lines from 2950 code files in BE-PUM, except comment lines and blank lines, are collected with duplicate removal and excessive white-space removal, resulting in a total of 30,402 code lines. We exclude all of the code lines that are in the code files emulating the target instructions implemented by human developers.

Collection of Explored Code Lines from Class Diagram

The exploration to generate potential code lines from the class diagram include 3 types (taking the class diagram fragment in Figure 2.5 as an example):

- Generating class declaration statement, e.g:
  `public class aaa extends X86InstructionStub {`
- Generating variable and method declaration and initialization statement, e.g:
  `public int opSize;`,
  `public X86InstructionStub();`,
  `BPState bPState = new BPState(Environment,AbsoluteAddress,Instruction);`
- Generating chained method calls based on object's datatype, e.g:
  `env.clone().getMemory().getStack().length();`
  `curState.getEnvironement().getMemory().getStack().equals(Stack);`
  We limit the number of next hops in a chained method call to be 5 - that is to only explore at max 5 more chained methods for a given variable of non-void datatype.

The total number of explored code lines with duplicate removal is 80,242.

The two collection scenarios results in a total of 110,644 code lines. These 110,644 code lines are to be stored in a database indexed by their semantics represented as vector values computed by a code embedding model - so-called a vector database. The following sections continues this process.

5.6.2 Code Embedding with CodeBERT

We measure the similarity between the queried code line and the code lines stored in the database by computing the cosine distance between their embedding vectors. We choose CodeBERT as the embedding model. Since tokens in code are just mnemonics or abbreviations and typically shorter than those in natural language, embedding code based solely on individual lines may lead to inaccurate results. It is common practice for code embedding models to require natural language descriptions alongside each code snippet to form a complete input. Therefore, we equip each line of the collected code lines with a text description. To obtain the text description, we ask ChatGPT-3.5-Turbo to give a brief summary (maximum 256 tokens) of the purpose of each code lines.

The usage of ChatGPT-3.5-Turbo is done via API provided by OpenAI [5]. Below is the Python code snippet used to obtain the summary:

```
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {
          "role": "system",
          "content": "You are a code summarizer."
        },
        {
          "role": "user",
          "content": f"Summarize this line of code:\n{code_line}"
        }
    ],
    temperature=1,
    max_tokens=256,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0
)
```

[5] https://platform.openai.com/docs/overview

Before resulting in choosing ChatGPT model for the summarization, we also try with a local model which is CodeLlama-2-34b-Instruct (CodeLlama-2, Instruct variant, 34 billion parameters). As we wish to limit the response length of CodeLlama (to under 50 words), we try with two types of prompting:

- Not explicitly informing the model about the word limit in the prompt but instead, set it in generation's hyper-parameters (case WLimHparam: Word Limit in Hyper-parameters).

```
<s>[INST] <<SYS>>
You are a code summarizer.
Summarize this line of code:

<</SYS>>

<code-line> [/INST]
```

- Explicitly inform the model about the word limit in the prompt (and also set it in hyper-parameters) (case WLimPrompt: Word Limit in Prompt).

```
<s>[INST] <<SYS>>
You are a code summarizer.
Under 50 words, summarize this line of code:

<</SYS>>

<code-line> [/INST]
```

Table 5.1 and 5.2 list out several samples demonstrating the results of BE-PUM's code lines and their obtained text description from ChatGPT-3.5-Turbo and the two cases of CodeLlama-2-34b-Instruct, respectively. The text in color red indicates that the response is not complete and gets cut off due to exceeding maximum length. As can be seen from the two tables, ChatGPT's response is more consistent, concise and shorter than CodeLlama's despite the configuration for ChatGPT indicates that maximum number of tokens is 256. As for CodeLlama responses, case *WLimPrompt* is more likely to produce responses that are within the specified length than *WLimHparam*. However, it is not always guaranteed. Another reason that we rely on ChatGPT for the automatic summarization is that the time it takes for running CodeLlama locally for this task is roughly 50 seconds per request while using OpenAI API for ChatGPT-3.5-Turbo returns each response in approximately 18 seconds with reasonable pricing rate.

For each pair of BE-PUM code line and its text description, following the data format for CodeBERT, we use the below template to form a training data-point:

```
    <s> {{text_description}} </s> {{code_line}} </s>
```

Following the standard fine-tuning process for a language model, we then obtain the fine-tuned CodeBERT on BE-PUM code base.

5.6.3 Vector Database Construction

We use ChromaDB [6] as our local database and semantic search engine. To populate the database, we simply use the fine-tuned CodeBERT model to get embeddings of all

[6] https://docs.trychroma.com

Table 5.1: Samples of code description obtained from ChatGPT-3.5-Turbo.

ID	BE-PUM's Code Line	ChatGPT-3.5-Turbo
1	env.getFlag().getDFlag()	Calls the getFlag() method from the env object and then calls the getDFlag() method on the result.
2	env.getMemory().clone(). getRegister().and(String,Value);	Gets the memory from the environment, creates a clone of it, accesses the register from the clone, and performs a bitwise AND operation with the given String value.
3	BPState bPState = new BPState(Environment, AbsoluteAddress,Instruction);	Creates a new object of type BPState with parameters Environment, AbsoluteAddress, and Instruction, and assigns it to the variable bPState.
4	Register register = new Register();	Creates a new instance of a class called Register and assigns it to a variable named register.
5	env.getSystem().getFileHandle(). moveFile(String,String);	Gets the system environment, accesses the file handle within the system, and invokes the moveFile method with two String parameters.
6	state.getEnvironement().getFlag(). changeFlagWithSUB(Value, Value, Environment, int);	Calls a method to change a flag in the environment by subtracting a specified value with the given parameters.

the collected BE-PUM code lines. The database then stores these code lines, indexes them by their vector embedding values.

To look up the top-k most similar BE-PUM code lines that potentially express the same idea as the Code Writer's generated code line in the query, we first use the fine-tuned CodeBERT to obtain the embedding of the generated code line. Then by computing cosine distance between each embedding vector stored in the database and the queried embedding, we fetch out the BE-PUM code lines whose embeddings are in the top-k closest to the queried. The cosine distance formula is

$$d = 1.0 - \frac{\sum (A_i \times B_i)}{\sqrt{\sum A_i^2}\sqrt{\sum B_i^2}}$$

where A and B are two vectors and i is an index into a vector's components.

Table 5.2: Samples of code description obtained from CodeLlama-2-34b-Instruct.

ID	CodeLlama-2-34b-Instruct (WLimPrompt)	CodeLlama-2-34b-Instruct (WLimHparam)
1	This line of code gets the DFlag value from the Flag object returned by the getFlag() method of the env object.	This line of code is accessing the DFlag property of an object stored in the env variable. The env variable is likely an environment object that contains various properties and methods. The getFlag() method is being called
2	This line of code is cloning the current memory state of the environment and performing a bitwise AND operation on the register and a string value.	The provided line of code appears to be a method call to the clone() method of a Memory object, and then a call to the and() method of a Register object. The method call is being made on the
3	Creating a new instance of the BPState class with the specified parameters Environment, AbsoluteAddress, and Instruction.	The code defines a new instance of the BPState class named bPState and passes three parameters to its constructor: Environment, AbsoluteAddress, and Instruction.
4	Declares a variable register of type Register and assigns a new Register object to it.	This line of code creates a new instance of the Register class and assigns it to a variable called register.
5	This line of code is using the env class's getSystem() method to get a reference to the system, and then using the getFileHandle() method to get a reference to the file handle. The moveFile	The code is calling the moveFile method of an object in the getSystem().getFileHandle() chain, passing in two string arguments. The method is used to move a file from one location to another.
6	This line of code is using the Environment class and its changeFlagWithSUB method to modify the value of a flag. The SUB parameter is used to specify the new value for the flag, and the Value	This line of code is using the changeFlagWithSUB method of the Flag class, which is part of the Environment class. The method takes four parameters: Value, Value, Environment, and int

Chapter 6

Experiments and Results

6.1 Experiment Setup

Hardware environment

During our selection of hardware environment, we observe that different GPU models produces different generated codes. Therefore, we includes two choices of environments that are available to us which includes: a single GPU A100 (40960MB, with DELL PowerEdge R750 Server) and a single GPU A40 (46068MB, with DELL PowerEdge R750 Server).

Software environment

As the aforementioned, the platform and library we use are PyTorch and HuggingFace's specialized libraries for transformer research. Other software products include ChromaDB for hosting a local search engine, OpenAI's ChatGPT-3.5-Turbo APIs for BE-PUM knowledge base's data preparation and Java Development Kit that consists of Java compiler for obtaining compiler's feedback. The training environments' operating system are Ubuntu, which is incompatible with the target code project BE-PUM. Since BE-PUM is built on Windows only, we need to establish a TCP connection to transfer generated code files to the BE-PUM project hosted on a Windows machine.

Experiment scenarios

Based on the main frame of the proposed system CoDeb, we empirically experiment it with different scenarios, with each is referred to one experiment. The naming of these experiments are:

- *Default-0* (CoDeb-Default-0): All hyper-parameters are set as the above mentioned in Chapter 5. The system runs on GPU A100, consists of two distinct generative models and finishes the first epoch of training.
- *Default-1* (CoDeb-Default-1): The same model in CoDeb-Default-0 finishes the second epoch of training.
- *NoRL* (CoDeb-NoRL): all configurations are similar to CoDeb-Defautl-0 but no training is applied.

- *A40* (CoDeb-A40): all configurations are similar to CoDeb-Defautl-0 but runs on GPU A40.
- (CoDeb-Single): all configurations are similar to CoDeb-Defautl-0 except that instead of two distinct generative models, we only use one model for both code writing and debugging tasks.

All the experiments undergo up to 3 iterations of Initial Code Generation and a maximum of 3 iterations of Iterative Code Generation.

6.2 Datasets

As described in Section 5.2.1, the text data used as input into the system includes natural language description of the x86 instruction and a java code template with respect to such instruction. Additionally, BE-PUM project-level context is used for BEPUM-KB component. The statistics for these datasets are:

- The natural language description of each instruction is extracted from the specifications that has been described in Section 2.1.2 of the Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A. As the manual comes with both 32-bit and 64-bit modes, the total number of instructions (1220 instructions) includes both 32-bit and 64-bit instructions. If we were to consider each variation of an instruction (a combination of Opcode and Operands) to be an independent instruction, then the total number is estimated to be 8274 instructions. For example, instruction called *CMOVcc* - Conditional Move has 90 variations such as *CMOVA* - Move if above, *CMOVNL* - Move if not less, and so on. In our case, we limit the number of instructions to 1147, neglecting several instructions for 64-bit only and expanding to a number of variations of the *CMOVcc* instruction that are used within BE-PUM. Particularly, we primarily focus on the existing set of 200 instructions including 120 previously implemented instructions in BE-PUM and the first 80 instructions among those 1147, ordered alphabetically.
- 200 java code templates, with each corresponding to a selected x86 instruction, are prepared using the method described in Section 5.2.1.
- 110,644 code lines of BE-PUM project-level context that is used specifically for BEPUM-KB, collected by the method described in Section 5.6.1.

6.3 Validation Metrics

Targeting at code compilation results, we use these two types of measurements: 1) Performance measurement that uses each generation time step as its counting unit and 2) Performance measurement that uses each data point in the dataset as its counting unit.

For measurement numbered 1), we use the following criteria:

- Static Syntactic Pass Ratio (SSPR):

$$\text{SSPR} = \frac{n_{\text{ss\_pass}}}{n_{\text{response}}}$$

The ratio shows the total number of times that the Code Writer produces a static-syntactically correct Java code file over the total number of times that the Code Writer responses. This criteria indicates how much the generative language model could follow the response format requirement given in the prompts (constrained decoding).

- Compiler Pass Ratio (CPR):

$$\text{CPR} = \frac{n_{\text{c\_pass}}}{n_{\text{response}}}$$

The ratio shows the total number of times that the Code Writer produces a static-syntactically correct Java code file that passes the compiler without an error over the total number of times that the Code Writer responses. This criteria indicates the capability of the system to generate project-level compilable code files.

- First-Error Improvement Ratio (FEIR):

$$\text{FEIR} = \frac{n_{\text{fei}}}{n_{\text{fixing\_response}}}$$

Where n_{fei} is the total number of times that the Code Writer produces a corrected version of the erroneous code file where the first error that previously occurred gets fixed and $n_{\text{fixing\_response}}$ is the total number of times that the Code Writer fixes an erroneous code file in Iterative code generation stage. This criteria assesses the effectiveness of using the Code Debugger to provide code fixing guidance and the ability of the Code Writer to follow such guidance in automatic code correction. On the other hand, the total number of errors is disregarded because fixing the first encountered error in the error list could mismatch the subsequent usage of the fixed code line such that the total number of errors may decrease or increase.

- First-Error Deterioration Ratio (FEDR):

$$\text{FEDR} = \frac{n_{\text{fed}}}{n_{\text{fixing\_response}}}$$

Where n_{fed} is the total number of times that the Code Writer produces a corrected version of the erroneous code file where the first error in the list turns out to occur earlier than in the previous version. The first errors from both times are not necessarily the same. This criteria also assesses the effectiveness of the code debugging and correcting task in CoDeb.

For measurement numbered 2), we use the following criteria:

- Static-Syntactically Correct Instructions Ratio (SSCIR):

$$\text{SSCIR} = \frac{n_{\text{ssc\_ins}}}{n_{\text{all\_ins}}}$$

where $n_{\text{ssc\_ins}}$ is the total number of instructions that have at least one static-syntactically correct Java implementation and $n_{\text{all\_ins}}$ is the total number of instructions.

- Compilable Instructions Ratio (CIR):

$$\text{CIR} = \frac{n_{\text{c\_ins}}}{n_{\text{all\_ins}}}$$

  where $n_{\text{c\_ins}}$ is the total number of instructions whose Java implementation passes project-level compilation in BE-PUM. These instructions can be then ready for semantic verification which is not in the scope of our current work.

Note that the Result section below reports these metrics in percentage.

6.4 Results

There are two major types of results with respect to static-syntactical correctness and project-level compilation correctness.

After the Static Syntactic Checking step in the Reward Function, the static-syntactical correctness is reported to demonstrate how well the generative models adhere to the specified response format for constrained decoding. It also verifies that the generated code passes static syntactic checks within the file-level context.

After the Compiler Checking step in the Reward Function, the project-level compilation correctness is report to show the end result of the system which is to achieve compilable code implementation.

For each type, overall results are presented first, followed by a more detailed analysis. Furthermore, Appendix B documents several demonstration of results.

6.4.1 Static-syntactical Correctness

Overall results

Table 6.1 reports the overall results of Static Syntactic Checking over the total number of Code Writer's responses (SSPR) and over the number of input instructions (SSCIR).

From Table 6.1, it can be seen that Default-0 scores the highest on both metrics SSPR and SSCIR, which means among the times that the Code Writer attempts to write a Java code file for all 200 selection instructions, 51.22% of such times it successfully produces a static-syntactically correct code. Among those 200 input instructions, there are 178 instructions (89%) having at least 1 static-syntactically correct implementation. Although NoRL has the least SSPR, it comes in second place in SSCIR metric. Further observations and explanation are documented next.

Table 6.1: Results on SSPR and SSCIR metrics.

Index	Experiment	SSPR	SSCIR
1	Default-0	**51.22**	**89**
2	Default-1	42.17	71.5
3	NoRL	37.32	79.5
4	A40	41.24	76
6	Single	42.54	75

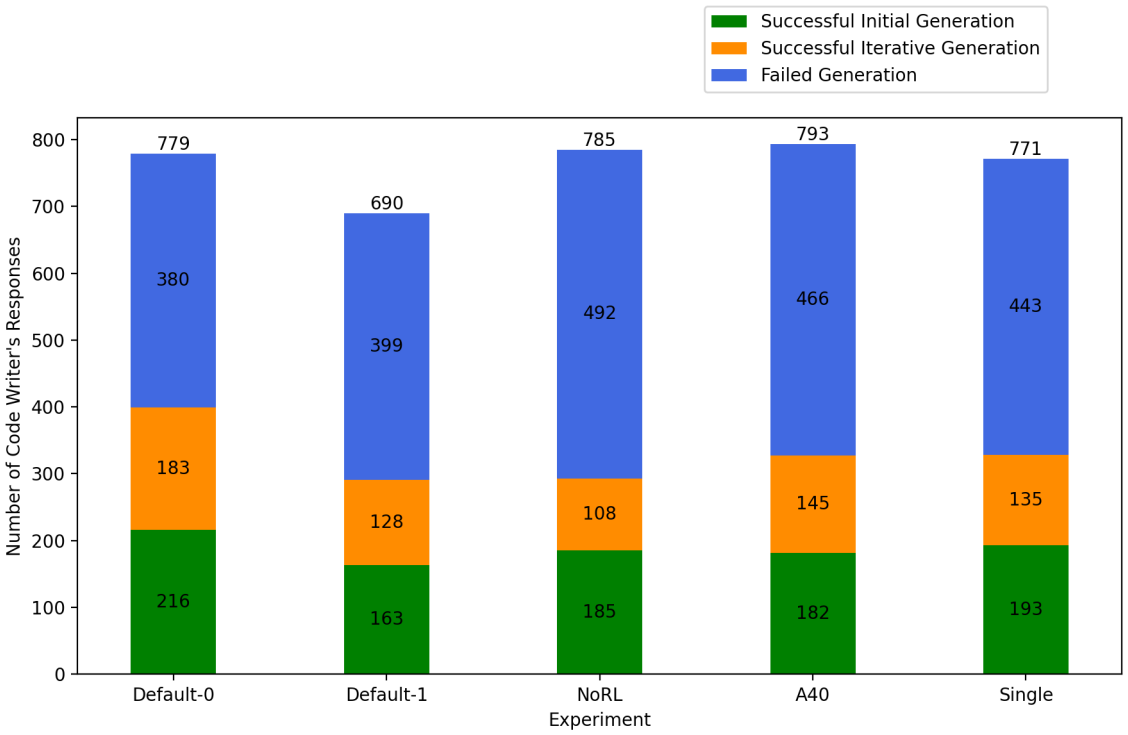


Figure 6.1: Result of Static Syntactically Checking for Initial Code Generation and Iterative Code Generation.

Result analysis

The result of static-syntactically correct and incorrect generation per Code Writer's response in shown in Figure 6.1. According to Chapter 5, there are two scenarios where the Code Writer produces a Java code file. In the Initial Code Generation scenario, across all experiments, it is approximately 24.58% of the total responses that a static-syntactically correct Java code file is produced. Meanwhile, the Iterative Code Generation with debugging process achieves an average of 18.32%. Because the total number of times that a static-syntactically correct Java code file is generated accounts for approximately 42.90% of all Code Writer's responses, we can see that it is hard to achieve constrained generation while still encouraging certain amount of randomness. Hence, the Code Writer needs to redo some more trials, resulting in a larger number of total responses per experiment. Considering the worst case where each of the 200 instructions need all 3 rounds of initial generation and then 3 rounds of iterative correction with debugging, the total number of responses that the Code Write must make in this case is 1800 ($200 \times 3 \times 3$), our experiments average to just 42.42% of such case.

For each individual experiment:

- Default-0: The experiment produces the highest number of static-syntactically correct Java code files (on response unit) while not requiring the most responses.
- Default-1: Compared to its previous epoch, in the second epoch the default-configured system takes less number of responses while having performance comparable to other experiments. This shows that it can obtain a static-syntactically correct Java code file earlier than other experiments. However,

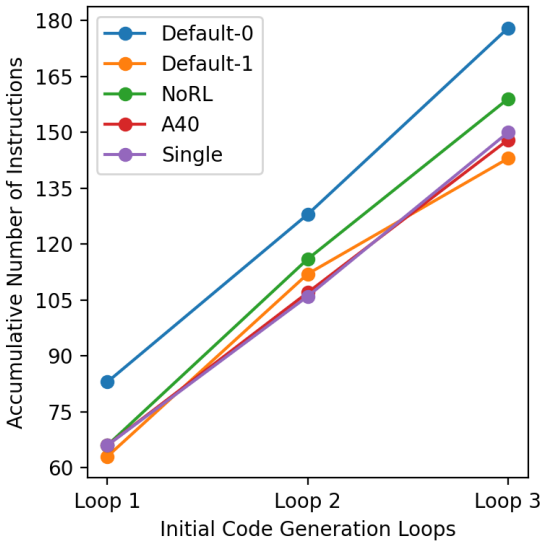


Figure 6.2: Accumulative number of static-syntactically correct instructions over Initial Code Generation's loops.

the number of instructions that have at least one static-syntactically correct implementation decreases. This drastic drop is caused by the fine-tuning process which is discussed more in Chapter 7.

- NoRL: From Figure 6.1 and Table 6.1, it can be seen that the experiment CoDeb-NoRL that does not undergo reinforcement fine-tuning produces the least static-syntactically correct Java files in response units (only 37.32% for SSPR). This is expected as the system is not governed by response format rules. However, it is still able to produce correct Java implementation for upto 79.50% of 200 x86 instructions, which hints the trade-off between syntactic and semantic tuning.
- A40: Experiment A40 needs the most responses to produce static-syntactically correct Java files while having moderate successful generation. The result shows that the generated part in the answer of A40 is often shorter than the answer produced when using GPU A100, an example is shown in Figure 6.3. Its correctness rate per instruction (SSCIR) is not as high as its counterpart running on GPU A100. However, its performance on SSPR and SSCIR are comparable to the Single experiment's.
- Single: The experiment requires as many responses as the Default-0 while producing less correct Java code files in both response units and instruction units. This suggests that separate generative models are more favorable for tasks that require distinct response formats.

Additionally, Figure 6.2 illustrates the increase in the total number of static-syntactically correct instructions across the three loops of Initial Code Generation. It can be seen that the total number of successful instructions increase steadily over the loops with Default-0 taking the lead. Default-1, A40 and Single's growths are moderate, while the increase in NoRL is more significant toward loop numbered 3.

If the Code Writer were limited to only one attempt, the count of static-syntactically correct instructions would be much lower, as demonstrated by the results of the first

loop in the figure. In fact, a human coder (without the aid of code linting) also attempts several times before obtaining a correct program. Hence, it is reasonable to let the Code Writer performs several trials if the previous attempt is not correct, and improve it via reinforcement reward through static syntactic checking. The figure demonstrates that allowing the Code Writer to have multiple attempts results in a higher number of instructions with static-syntactically correct implementations.

6.4.2 Project-level Compilation Correctness

The overall results shown here include statistics on the number of generated code files that successfully compile at the project level within the target project BE-PUM. Following this, the analysis presents how the compiler's feedback, interpreted by the Code Debugger, impacts the correction of the code files generated by the Code Writer.

Overall result

Table 6.2: Results on CPR and CIR metrics.

Index	Experiment	CPR	CIR
1	Default-0	**2.57**	**10**
2	Default-1	1.16	4
3	NoRL	1.66	6.5
4	A40	1.01	4
6	Single	1.95	7.5

Table 6.2 reports the overall results on two metrics CIR and CPR in percentages. The CIR column indicates the percentage of instructions out of the selected 200 that are successfully compiled in BE-PUM. Meanwhile, the CPR column displays the percentage of compilable responses out of all the Code Writer's responses. As can be seen that Default-0 scores highest on both measurements. The following-ups are Single and NoRL. The reason why these two experiments have more significant results than Default-1 and A40 can be explained as follows:

- Default-0: A subset of pre-trained weight matrices are tuned in favor of producing compilable codes, which may affect the output semantics but not significant on the first epoch.
- NoRL: The pre-trained weight matrices are unaltered, therefore the pre-trained semantics is preserved. However, as output syntax is not constrained, the number of syntactically correct generation is few, and thus the final number of compilable codes is less than Default-0.
- A40: Similar to Default-0, a subset of pre-trained weight matrices are tuned in favor of producing compilable codes. However, H100 GPU may advance more than A40 GPU in terms of floating-point precision. Hence, the output probabilities for decoding tokens are different. Figure 6.3 also demonstrates such difference.

Additionally, Figure 6.4 summarizes the number of compilable generated code files of the system through two stages: Initial Code Generation and Iterative Code Generation. It can be seen from the figure that experiment Default-0 obtains the most compilable files, Single and NoRL come in second and the remaining experiments

```
        // Generate from here

        // Set flag value
        if (eax.getVal() > 9) {
            AFlag.setVal(true);
        } else {
            AFlag.setVal(false);
        }

        // Calculate carry
        if (eax.getVal() > 255) {
            CFlag.setVal(true);
        } else {
            CFlag.setVal(false);
        }

        // Modify register value
        eax.setVal(eax.getVal() & 0xFF);

        // Update state
        curState.setRegister(eax);

        return curState;
    }
}
```

(a) Experiment A40.

```
        // Generate from here
        LongValue al = (LongValue) env.getRegister().getRegisterValue("al");
        LongValue ah = (LongValue) env.getRegister().getRegisterValue("ah");

        // Adjust the AL register to contain the correct 1-digit unpacked BCD result
        LongValue result = new LongValue(0);
        if (CFlag.getValue()) {
            result = al.getValue() + 1;
        } else {
            result = al.getValue();
        }
        al.setValue(result);

        // Set CF and AF flags
        if (CFlag.getValue()) {
            CFlag.setValue(true);
            AFlag.setValue(true);
        } else {
            CFlag.setValue(false);
            AFlag.setValue(false);
        }

        // Clear bits 4 through 7 of the AL register
        al.setValue(al.getValue() & 0b00001111);

        // Update AH register
        ah.setValue((al.getValue() & 0b10000000) >> 7);

        // Return updated state
        return curState;
    }
}
```

(b) Experiment Default-0 with GPU A100.

Figure 6.3: Example on generation differences between GPU A40 and GPU A100: *the generative models are given the same prompt asking for Java implementation for instruction AAA, in the first loop of Initial Code Generation.*

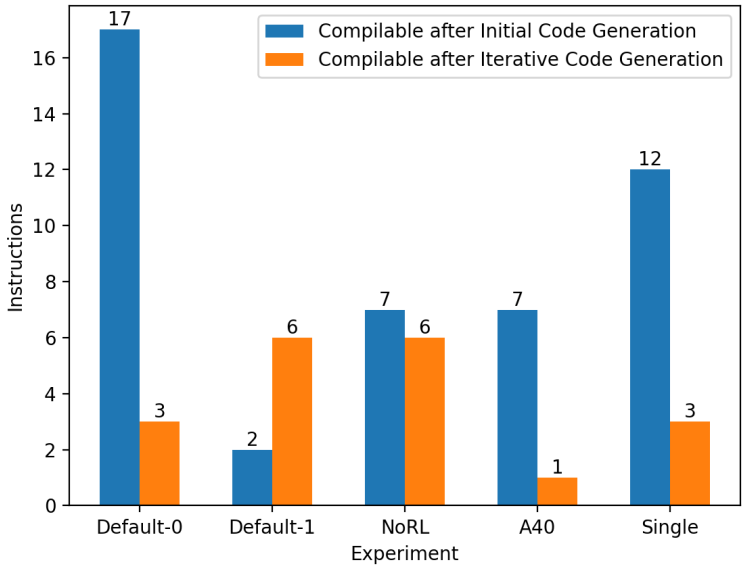


Figure 6.4: Number of successfully compiled generated code files in BE-PUM.

obtains moderate results. While the compilable files obtained mostly from the Initial Code Generation for all experiments except Default-1, for Default-1, the compilable files obtained after debugging is higher than those in the initial generation. This can be explained by the effect of reinforcement fine-tuning using only feedback on the correctness of syntax without considering semantic feedback. Nonetheless, a rise in the compilable files due to debugging demonstrates the effective role of the Code Debugger.

Result analysis

The results of utilizing feedback from the compiler to fix errors in the generated code files is shown in Figure 6.5. The figure reports the total number of times that the Code Writer produces corrected code with Improvement, Deterioration or Unchanging status, while Table 6.3 presents it in percentages. These labels are furthered explained as follows:

- Improvement: Indicates that after receiving consultation from the Code Debugger, the Code Writer produces a static-syntactically correct code file, in which the first error that occurred in the previous compilation is now fixed.
- Deterioration: Indicates that after receiving consultation from the Code Debugger, the Code Writer produces a static-syntactically correct code file, in which the first error that occurred in the previous compilation may not yet be fixed but another error now occurs earlier than the previously found one. This shows that the code correction wrongly changes the correct code lines.
- Unchanging: Indicates that after receiving consultation from the Code Debugger, the Code Writer produces a static-syntactically correct code file, in which nothing has been modified.

Among these three label, Improvement indicates a good code fix, which is favorable. Deterioration indicates a bad fix, Unchanging indicates an ineffective fix, which are

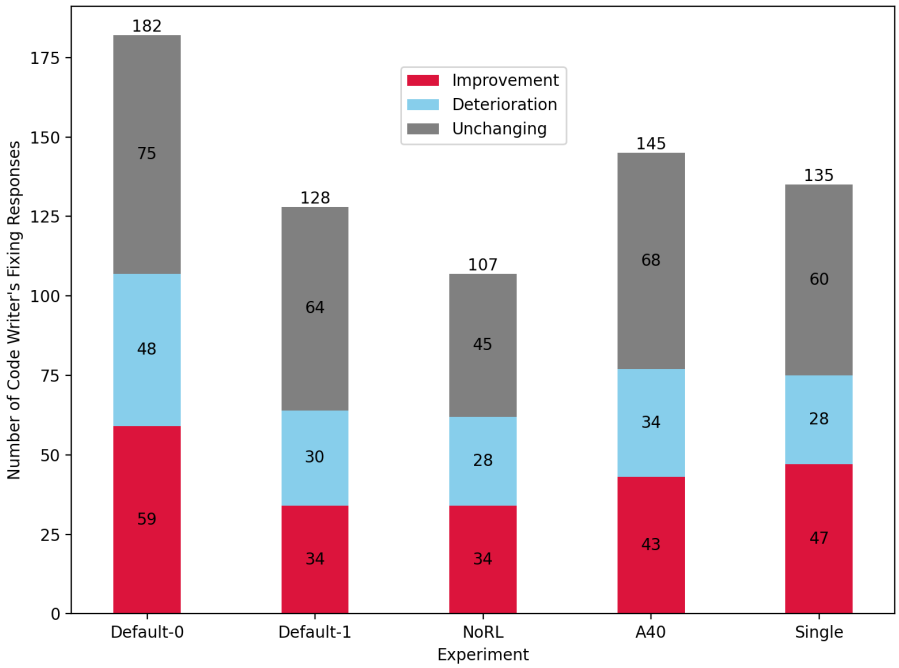


Figure 6.5: Effectiveness of code correction in Iterative Code Generation.

both undesired.

Table 6.3: Results on FEIR and FEDR metrics.

Index	Experiment	FEIR	FEDR
1	Default-0	32.42	26.37
2	Default-1	26.56	23.44
3	NoRL	31.78	26.17
4	A40	29.66	22.33
6	Single	**34.81**	**20.74**

Individual experiments: The result reported by Figure 6.5 and Table 6.3 shows that:

- Default-0: The Default-0 experiment corrects the most code files while having fewer Deteriorations than Improvements. However, the number of Deteriorations and Unchangings are also significantly higher than other experiments.
- Default-1: The Default-1 experiment's results are less than its previous epoch (Default-0) on all three indicators. Adding up with its result on static-syntactic correctness, the problems may have root in how reinforcement learning changes the generative models' parameters in the way that favors the conformance to response format and static-syntactical correctness than code coherence. This behaviour is further stated in Chapter 7.
- NoRL: The system when not using any fine-tuning is shown to obtain comparable results with the number of Improvements being higher than the Deteriorations, while the number of Unchanging code correction is the fewest. Still, taking its result in static-syntactical correctness into consideration, we can see that its Improvement number is just moderate while requiring as many responses as Default-0. Additionally, as being non fine-tuned, this result cannot be further improved like those of the experiments with fine-tuning.

- A40: Experiment A40 has moderate results with the number of Improvements being larger than its own Deteriorations and larger than which of Default-1.
- Single: Using only one generative model to iteratively modify and assess the same code file shows certain benefit since the result is also moderate like which of the two-model system in experiment A40. Percentage results from Table 6.3 also shows that this experiment scores the highest in FEIR and the lowest in FEDR. Adding up to the fact that this needs a significant number of responses, we can infer that most of the times the system in this experiment produces static-syntactically incorrect code files. But when it manages to produce it correctly, it has higher chance of obtain an Improvement. Compared to the experiment Default-0 with two distinct models, we can see that Default-0 outperforms it while requiring similar amount of total responses.

Assessing Improvement over total responses leads to a hypothetical estimation discussed in the following content: It can be seen that the total sum of all indicators presented in Figure 6.5 here also indicates the number of static-syntactically correct responses the Code Writer makes in Iterative Code Generation (see Figure 6.1, a small difference of 1 is due to a log file from Javac that cannot be decoded). If we assume the static-syntactically correct responses in Figure 6.1 of Initial Code Generation is obtained immediately on the first try, then the remaining Failed Generation is considered to have occurred in Iterative Code Generation. Table 6.4 estimates this hypothetical situation where the values ($= \frac{\#\text{Failed Generation}}{\#Improvement}$) means the average redo times that the Code Writer must do in order to obtain one Improvement. Hence, a smaller value is more favorable. It can be seen from the table that Default-0 has the lowest trials to obtain an Improvement, Default-1 and Single needs a moderate number of redos while NoRL requires the most trials and its total number of Improvement is among the lowest.

Table 6.4: Estimated number of the Code Writer's responses needed for obtaining one Improvement in Iterative Code Generation.

Experiment	Default-0	Default-1	NoRL	A40	Single
Estimated responses	**6.67**	10.23	12.62	12.59	10.55

Individual loops: This result presents how Improvement and Deterioration changes throughout each debug loop in Iterative Code Generation of each experiment. Figure 6.6 presents the said result.

Note that an Improvement (as well as Deterioration and Unchanging) is counted between two subsequent code correction turns that both pass compiler checking. For example, an Improvement happens after loop #3 of Iterative Code Generation, this means that if loop #2 produces a static-syntactically correct code file, then the comparison is between the two compiled code files of the said loops. Else, if loop #2 does not produce such static-syntactically correct code file, then the comparison is meant for previous loops' generated code, e.g: loop #1, if it has a static-syntactically correct code file; or the code file from the Initial Code Generation, which is guaranteed.

It can be seen from the Figure 6.6 that the highest number of Improvements and Deteriorations both occur in debug iteration 1. The system in experiments A40 and Default-0 achieves the highest Improvement in the first debug loop, while Single obtains a moderate level of Improvement, and Default-1 and NoRL are among the

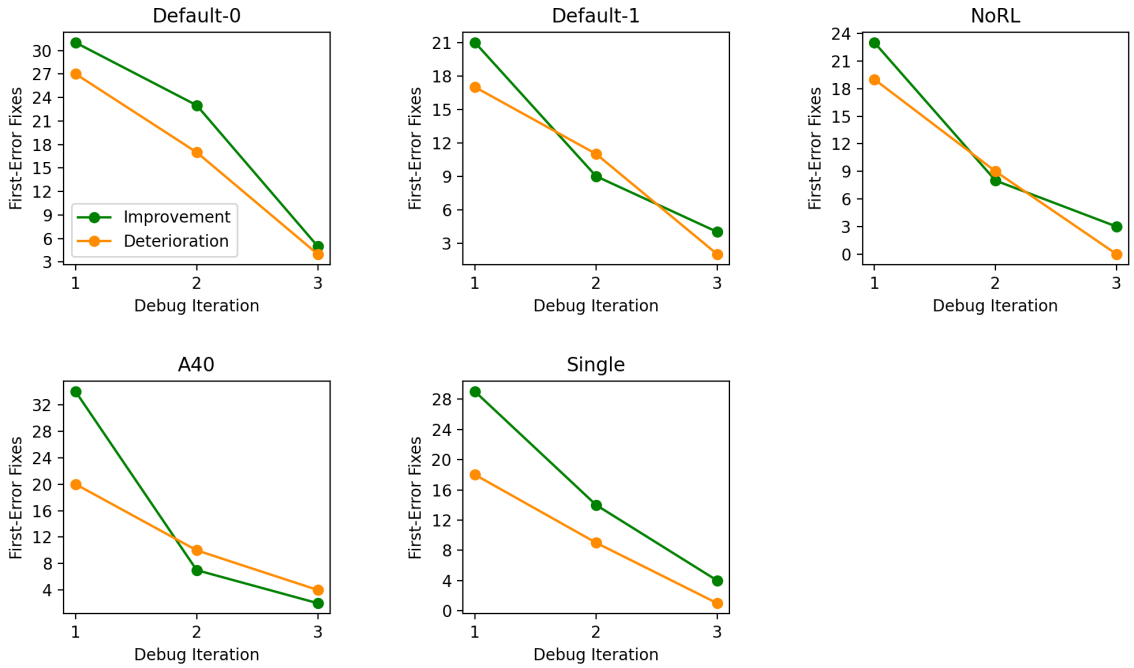


Figure 6.6: Number of Improvements and Deteriorations per debug iteration.

lowest.

Although an Improvement occurs in previous debug loop, the system makes no guarantee that such Improvement is retained in the subsequent debug loop. Hence, the Improvements and Deteriorations decrease over debug iterations is explained as follows:

- The Code Writer does not provide answers that are static-syntactically correct after the first debug loop.
- The Code Writer in the later loops refrains from making changes to the given erroneous code file, resulting in an Unchanging status.
- After a code line is correctly fixed, the number of errors may increase due to changes made in that line (e.g., changing data types, variable names, method names, etc.). Subsequent attempts to fix these errors may result in success, failure, or no change with success being less likely to happen (decrease of Improvement is steeper than Deterioration's).

In conclusion, the fact that the number of Improvements, especially in the first debug iteration is larger than the number of Deteriorations shows that the system has the ability to self-correct its generated code files. However, as the system does not have a mechanism to retain such Improvements in the subsequent debugging iterations, the number of static-syntactically valid changes (Improvements and Deteriorations) decreases with the steepest decline occurring in the number of Improvements.

6.4.3 Supplementary Results

Removal of Debugging Step

This experiment is carried out to determine the impact of performing debugging step in code correction. In the default system, we extend the code correction task with

an intermediate step of error explanation and solution suggestion done by component Code Debugger. This experiment removes such step and directly asks the Code Writer to fix the generated code given the raw feedback text from the compiler. Thus, the prompt now used for Iterative Code Generation by the Code Writer is:

```
You are an expert Java programmer.
Your job is to fix an erroneous program given compiler error message.
The program that needs correction and the error message are in the input.
Use suggestion in the context.
Output the corrected version of the given program only.
Do not explain anything before or after the program.
Your answer should start with ‘‘‘java mark and end with ‘‘‘ mark.
Provide comment for each block of codes.

### Input:
- The erroneous program <class-name>.java:
<error-code>

- The compiler error message:
<error>

### Context:
Suggest using these snippets:
<suggest-snippets>

### Response:
```

This experiment result shows the inferior to the chain-of-thought approach of the original design. The total number of compilable instructions out of 200 is only 5 (CMOVNB, CMOVNGE, DAA, NOP, and STD), which results in 2.5% on CIR metric. These compilable instructions are all resulted from Initial Code Generation. It is seen that most of the time the response returned by the Code Writer is instead the explanation of why the error happens or just a fragment of corrected code mixed in with natural language. There are cases that the Code Writer actually produces static-syntactically correct code for this type of prompt, but throughout several iterations of code correction, none has succeeded.

Running Default-0 on a full-length dataset

Due to time and resource constraints, only the best-effort experiment Default-0 completed the entire dataset of 1147 selected instructions. Out of these, Default-0 produced 1023 instructions with static-syntactically correct implementations. However, only 165 of these instructions successfully passed compiler verification in BE-PUM, representing 14.39% of the full dataset.

Comparison with ChatGPT

As a point of reference, we asks ChatGPT-3.5-Turbo to generate Java implementation for the said 1147 instructions. The input prompt of ChatGPT we use is different from our system: in addition to the description of an instruction and the prepared Java template, ChatGPT also receives the whole class diagram of the current code base in BE-PUM. With that, the model achieves 94.97% instructions with static-syntactically correct implementations. Among which, there are 340 instructions whose implementations pass verification by the compiler in BE-PUM project. The exact number of parameters in ChatGPT-3.5-Turbo is undisclosed, but it is anticipated to be greater than 7 billion. Therefore, its success rate, which is double that of our local model, is expected. This shows that our results are reasonable for a local solution.

Table 6.5: CodeBLEU between Default-0 and semi-automatic approach.

Ngram match score	**Weighted ngram match score**	**Syntax match score**	**Dataflow match score**	**CodeBLEU**
10.43	26.06	54.51	70.6	40.4

CodeBLEU score of Default-0

In comparison to the previous semi-automatic instruction emulation in BE-PUM, we also conduct an assessment between Default-0 and the method using the CodeBLEU metric [66]. Since the same code semantics can be expressed through different implementations, this assessment serves only as a reference, demonstrating how the fully generated code is able to correctly reuse existing code from BE-PUM, including variable names and function calls. Table 6.5 reports the measurement in percentage. We can see that the n-gram matches between the two implementation is low, while syntax match is moderate and dataflow match is significantly higher. The overall values averages to a BLEU score of 40.4%.

Chapter 7

Discussion

7.1 Feasibility of CoDeb System

The non-fine-tuning experiment (NoRL), although achieving reasonable results, requires a large number of attempts to obtain them. The fact that these results are not the best, and the generative components cannot be further improved, highlights the importance of governing and enhancing generation through reinforcement fine-tuning.

By fine-tuning via reinforcement learning with feedback from software engineering tools, the system does not require domain-specific supervised labelled datasets. Rather, the Code Writer and Code Debugger explore the solution space themselves. In our experiment settings, we only set a fixed number of trials per code writing stage - Initial Code Generation and Iterative Code Generation. In practice, it is often required more trials in order to correct a coding error.

However, as evidenced by the experiment results, our system still has flaws. The following content is dedicated to discussing these issues.

7.2 Issue of Constrained Decoding

Our theme is an instance of generation problem that requires constrained decoding, particularly for syntactical constraint and contextual constraint. The results demonstrate the difficulty of consistently achieving constrained generation. Our system attempts to enforce those constraints on the output by specifying them in the input prompts and employing reinforcement fine-tuning to reward the obedience. However, it is evident that this approach is insufficient, which requires generative models to perform multiple trials in hopes of producing a correct result.

Another factor that adds up to the issue is the problem of sparse reward in reinforcement learning of our case. The reward used for the generative models is delayed in the sense that each of their actions which is to generate one token at a time, does not receive any reward signal until the entire sequence of actions is completed. Therefore, such way of giving reward is not informative enough to guide the generative model at each step in choosing output tokens. Besides, we do think that the impact of each criterion in our current Reward Function's design must be more thoroughly assessed.

In general, the survey paper [9] points out that the constrained decoding challenge

specifically comes from multiple hardships including incorporation of the constraints into a model's objective function during training; and lack of datasets, evaluation metrics and optimization methods. There have been multiple work that aims to achieve constrained decoding, especially for code-related tasks such as llama.cpp [1] which allows defining formal grammars to constrain model outputs, and Artificial Intelligence Controller Interface (AICI) project of Microsoft [67].

7.3 Trade-off between Ensuring Syntactical Constraint and Semantics

Our experiments Default-0, Default-1 and NoRL demonstrates consequences of tuning generative models towards syntactical adherence while neglecting semantics. It can be seen that the generative model in the later epoch is quicker to produces answers that are more adherent to the syntactical constraints. However, the code coherence, in turn, slowly degrades. This is evident as the results of experiment Default-1 are lower than those of Default-0 and NoRL.

The inefficiency of relying solely on syntactical constraints for project-level code generation emphasizes the importance of incorporating methods for semantic checking via testing. Therefore, it is recommended that using feedback from testing is the next step of this theme. Further phases in compilation process such as code optimization may also apply in order to help the Code Writer produce more efficient codes.

7.4 Issue of Preserving Improvement in Iterative Code Generation

There are two main reasons why the Iterative Code Generation is unstable in terms of preserving the past good fixes:

- State-less Code Debugging: Our system's code debugging is stateless - meaning that only the most recent version of the corrected code file is retained, with earlier versions being discarded. Therefore, penalty for repeating generation cannot be given, which results in a large number of Unchanging status. On the other hand, the reason for why the current design of the system is stateless is that concatenating history of code correction would result in a drastic increase in input size. Consequently, there may not be sufficient memory for training, and it could also lead to confusion for the generative model regarding which tokens it should prioritize.
- Non-monotonic behaviour of generative models: The generated answer of previous round is not guaranteed to be retained in the next round, which adds to the non-preservation of Improvement. This is due to these major reasons:
  - The content of the prompt may be not instructive enough or informative enough. In fact, constructing prompts is know-how knowledge. Therefore, it is hard to devise an optimal prompt for a given problem, rather only a sub-optimal prompt can be investigate.

[1] https://github.com/ggerganov/llama.cpp/tree/master

– There are non-monotonic processes in the construction of a deep learning models such as activation functions like Swish [68] and its variation SwiGLU [69] are non-monotonic. In fact, SwiGLU is used in CodeLlama models.

– Generative models rely on probabilistic sampling in its decoding stage to obtain an output sequence. One conceptual idea is to make tokens from previously correct version more probable to obtain in the next generation turn, however, this may restrict the creativity of the generative models to some degree.

7.5 Applicability of Chain-of-Thought Prompting

One of the reasons why we approach with almost-zero-shot prompting is to reduce the input size as text specification of an instruction can be large. Although this approach may be sufficient in terms of training resource usage for our scope, its output quality, however, can be underwhelming. An example for this is the failed case in Appendix B.2 where there is supposed to be no RF flag involved in the execution of instruction SAHF. One reason for this is that providing a full body of description text as input all at once may cause the generative components to attend to the wrong tokens as the context length is large and attention can be sparse, which leads to hallucinations (producing made-up, incorrect, or misleading results). Additionally, if the new input size is too large, out-of-memory error could still occur.

A natural approach for a human developer would be to sequentially process the specification part-by-part, which is reasonable due to property #4 of the specifications in ISA manuals, as stated in our problem statement (Chapter 1). For generative models, Chain-of-Thought prompting [70] is an effective technique to replicate this process. The idea is to prompt the model to generate a series of intermediate reasoning steps or thoughts leading to the final answer, rather than expecting it to produce the answer directly. As such, we can break down the input description into manageable chunks (typically sentences or short paragraphs) and let the models generate for each chunk, thus implement an instruction step by step. Although this approach may disrupt the continuity of context, a possible mitigation is to ensure that each step maintains its pre-condition and post-condition. Another viable benefit of this approach is that it should make it easier to address errors in the generated code, both syntactically and semantically.

Chapter 8

Conclusion

8.1 The Effectiveness of CoDeb System

It can be seen from the experiment results that the design of CoDeb system is effective and presents a promising approach. Specifically, in the system, code writing and debugging are done automatically via two generative components, functioning similarly to how two developers communicate to resolve code errors.

Instead of manually interfering in the debugging stage such as several work in the literature [13, 14], in CoDeb system we grant it to a generative component because nowadays code-specific generative models are equipped with debugging ability. Though human expertise is still superior for complex problems, having the task done automatically while making the model improvable is more desired towards the aim of automation and human effort reduction.

CoDeb is adaptable to new code domains. The generative components of CoDeb are replaceable while the code snippet knowledge base can be recreated with new content. Additionally, the software engineering tools used to give reward values can be substituted with other verification tools depending on the specific task.

8.2 The Limitation of CoDeb System

Within the scope of achieving project-level compilability, the main limitation of the system is that it lacks the ability to guarantee its outcome's correctness progressively. The root of the problem actually lie in the fact that deep learning models lacks of expressiveness - that is to explain the transformation of an input to a corresponding output within the model. If it were made clear, then the incorporation arbitrary constraints would be controllable and hence, the desired outcome could be assured. To sum up, the current limitations include:

- Constrained decoding is hard to guaranteed.
- Within the current scope of achieving compilability, ensuring only syntactical correctness while neglecting semantic verification may hurt performance.
- Improvement in code correction is not progressively retained.

8.3 Future Directions

Future work of our theme includes two major tasks as follows:

- **Improving the current results.**
  - As indicated in discussion, the next important step is to include semantic verification stage by further performing black-box checking, including dataflow analysis in compiler at higher optimization level, verification by test cases, and ultimately symbolic execution for full path coverage.
  - As incorporating more verification stages increases complexity, breaking down the input specifications into steps and applying Chain-of-Thought prompting would make the generation more manageable and debuggable.
  - To mitigate the issues of sparse rewards in reinforcement learning, a deeper study into reinforcement fine-tuning of generative models could assist in designing a more effective reward function.
  - Besides, to increase the efficiency, it is essential to enforce more effective methods to achieve more constrained decoding such as 1) Combating the sparsity of the reward by intervening directly at each step of the token exploration process in generative models; and 2) Considering about applying output template or grammar in a more systematic way.
  - Mechanisms for progressive code correction should be studied and implemented.
  - The next nearest target is to adapt the system to similar domains such as manual specifications for the 64-bit version and other processor architectures.
- **Generalizing to broader scenarios.** The research aims at automating tasks in software development, specifically focusing on code generation from natural language descriptions. Acknowledging the current situation, our work aims at leveraging Class Diagram of a software development project to explore the topic of generating a code project from scratch. The necessary sub-objectives are:
  1. Automatically generating class diagram from given software specification.
  2. Utilizing class diagram as parsable and retrievable knowledge for generative model to take reference from.
  3. Construct project-level text-to-code generation system that is capable of using such knowledge to build a code project from scratch and continuously.

  Among the three, sub-objectives numbered #2 and #3 are being investigated altogether in our current work.

Appendix A

Policy Gradient Theorem

The Policy Gradient Theorem is proved as follows:

$$
\begin{aligned}
\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= \frac{\partial v_\pi(s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\
&= \frac{\partial \sum_a \pi(a|s, \boldsymbol{\theta}) q_\pi(s, a, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \quad \text{(Eq. 4.6)} \\
&= \sum_a \left[\frac{\partial \pi(a|s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} q_\pi(s, a, \boldsymbol{\theta}) + \pi(a|s, \boldsymbol{\theta}) \frac{\partial q_\pi(s, a, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right] \quad \text{(Derivative product rule)} \\
&= \sum_a \left[\frac{\partial \pi(a|s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} q_\pi(s, a, \boldsymbol{\theta}) + \pi(a|s, \boldsymbol{\theta}) \frac{\partial \sum_{s', r} P(s', r|s, a)(r + v_\pi(s', \boldsymbol{\theta}))}{\partial \boldsymbol{\theta}} \right] \quad \text{(Eq. 4.7)} \\
&= \sum_a \left[\frac{\partial \pi(a|s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} q_\pi(s, a, \boldsymbol{\theta}) + \pi(a|s, \boldsymbol{\theta}) \sum_{s', r} P(s', r|s, a) \frac{\partial v_\pi(s', \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right] \\
&= \sum_a \left[\frac{\partial \pi(a|s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} q_\pi(s, a, \boldsymbol{\theta}) + \pi(a|s, \boldsymbol{\theta}) \sum_{s'} P(s'|s, a) \frac{\partial v_\pi(s', \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \right] \qquad \text{(A.1)}
\end{aligned}
$$

To further unroll the recursion, we first simplify the notation by changing partial derivative on $\boldsymbol{\theta}$ to $\nabla_{\boldsymbol{\theta}}(\cdot)$ and letting π implicitly denote a policy parameterized by $\boldsymbol{\theta}$. Supposed the the trajectory from $s^{(0)}$ to some terminal state $s^{(k)}$ involves transition $s^{(0)} \to s^{(1)} \to s^{(2)} \cdots \to s^{(k)}$, the Equation A.1 becomes:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} v_\pi(s^{(0)}) \\
&= \sum_a \left[\nabla_{\boldsymbol{\theta}} \pi(a|s^{(0)}) q_\pi(s^{(0)}, a) + \pi(a|s^{(0)}) \sum_{s^{(1)}} P(s^{(1)}|s^{(0)}, a) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(1)}) \right] \\
&= \sum_a \nabla_{\boldsymbol{\theta}} \pi(a|s^{(0)}) q_\pi(s^{(0)}, a) + \sum_a \left[\pi(a|s^{(0)}) \sum_{s^{(1)}} P(s^{(1)}|s^{(0)}, a) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(1)}) \right] \\
&= \sum_a \nabla_{\boldsymbol{\theta}} \pi(a|s^{(0)}) q_\pi(s^{(0)}, a) + \sum_{s^{(1)}} \sum_a \pi(a|s^{(0)}) P(s^{(1)}|s^{(0)}, a) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(1)})
\end{aligned}
\qquad \text{(A.2)}
$$

Denote the probability transitioning from state $s^{(i)}$ to $s^{(m)}$ through $m - i$ steps as $P(s^{(i)} \to s^{(m)}, m - i, \pi)$.

When $s^{(i)}$ and $s^{(m)}$ are the same state (e.g: taking 0 step at the initial state, $s^{(0)} \to s^{(0)}$), we have:

$$P(s^{(0)} \to s^{(0)}, m-i=0, \pi) = 1$$

When $s^{(m)}$ is a distinct subsequent state of $s^{(i)}$:

$$P(s^{(i)} \to s^{(m)}, m-i=1, \pi) = \sum_a \pi(a|s^{(i)}) P(s^{(m)}|s^{(i)}, a)$$

Recursively, the probability of transitioning from state $s^{(i)}$ to state $s^{(m)}$ with $m-i>1$ steps is:

$$P(s^{(i)} \to s^{(m)}, m-i>1, \pi) = \sum_{s^{(m-1)}} P(s^{(i)} \to s^{(m-1)}, (m-i)-1, \pi) P(s^{(m-1)} \to s^{(m)}, 1, \pi)$$

Let us also denote $A^{(i)} = \sum_a \nabla_{\boldsymbol{\theta}} \pi(a|s^{(i)}) q_\pi(s^{(i)}, a)$. The Equation A.2 then becomes:

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} v_\pi(s^{(0)}) \\
&= A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(1)}) \\
&= A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) \left[A^{(1)} + \sum_{s^{(2)}} P(s^{(1)} \to s^{(2)}, 1, \pi) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(2)}) \right] \\
&= A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) A^{(1)} \\
&\quad + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) \sum_{s^{(2)}} P(s^{(1)} \to s^{(2)}, 1, \pi) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(2)}) \\
&= A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) A^{(1)} \\
&\quad + \sum_{s^{(2)}} \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) P(s^{(1)} \to s^{(2)}, 1, \pi) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(2)}) \\
&= A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) A^{(1)} + \sum_{s^{(2)}} P(s^{(0)} \to s^{(2)}, 2, \pi) \nabla_{\boldsymbol{\theta}} v_\pi(s^{(2)}) \\
&= \sum_{s^{(0)}} P(s^{(0)} \to s^{(0)}, 0, \pi) A^{(0)} + \sum_{s^{(1)}} P(s^{(0)} \to s^{(1)}, 1, \pi) A^{(1)} + \\
&\quad + \sum_{s^{(2)}} P(s^{(0)} \to s^{(2)}, 2, \pi) A^{(2)} + \cdots + \sum_{s^{(k)}} P(s^{(0)} \to s^{(k)}, k, \pi) A^{(k)} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(since } \nabla_{\boldsymbol{\theta}} v_\pi(s^{(k)}) = 0\text{)} \\
&= \sum_{m=0}^{k} \sum_{s^{(m)}} P(s^{(0)} \to s^{(m)}, m, \pi) A^{(m)} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(A.3)}
\end{aligned}$$

Note that $s^{(m)} \in \mathcal{S}$ denotes iterating over all possible states in $\mathcal{S}$ at step m^{th}. With each step to a subsequent state acts as one sampling from $\mathcal{S}$, law of large numbers shows that when $k \to \infty$, the probability of transition $s^{(0)} \to s^{(m)}$ through $m \to k \to \infty$ steps converges to a true value, if exists. Applying the On-policy distribution in episodic tasks [1], we have: $\eta(s^{(\cdot)}, \pi) = \sum_{m=0}^{\infty} P(s^{(0)} \to s^{(\cdot)}, m, \pi)$. Equation A.3 then

[1] Chapter 9 of the same book in [58]

becomes:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} v_{\pi}(s^{(0)}) \\
&= \sum_{s^{(\cdot)} \in \mathcal{S}} \eta(s^{(\cdot)}, \pi) A^{(\cdot)} \\
&= \sum_{s' \in \mathcal{S}} \eta(s', \pi) \sum_{s^{(\cdot)} \in \mathcal{S}} \frac{\eta(s^{(\cdot)}, \pi)}{\sum_{s' \in \mathcal{S}} \eta(s', \pi)} A^{(\cdot)} \\
&= \sum_{s' \in \mathcal{S}} \eta(s', \pi) \sum_{s^{(\cdot)} \in \mathcal{S}} \mu(s^{(\cdot)}, \pi) A^{(\cdot)} \\
&\propto \sum_{s^{(\cdot)} \in \mathcal{S}} \mu(s^{(\cdot)}, \pi) A^{(\cdot)} \\
&= \sum_{s^{(\cdot)} \in \mathcal{S}} \mu(s^{(\cdot)}, \pi) \sum_{a \in \mathcal{A}} \nabla_{\boldsymbol{\theta}} \pi(a | s^{(\cdot)}) q_{\pi}(s^{(\cdot)}, a) \\
&= \sum_{s \in \mathcal{S}} \mu_{\pi}(s, \boldsymbol{\theta}) \sum_{a \in \mathcal{A}} q_{\pi}(s, a, \boldsymbol{\theta}) \frac{\partial \pi(a | s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \qquad \blacksquare
\end{aligned} \tag{A.4}
$$

The commonly used notation of $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ involves expectation notation, which can be obtained by further modifying Equation A.4 as follows:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \sum_{s \in \mathcal{S}} \mu_{\pi}(s, \boldsymbol{\theta}) \sum_{a \in \mathcal{A}} q_{\pi}(s, a, \boldsymbol{\theta}) \frac{\partial \pi(a | s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \\
&= \sum_{s \in \mathcal{S}} \mu_{\pi}(s, \boldsymbol{\theta}) \sum_{a \in \mathcal{A}} \pi(a | s, \boldsymbol{\theta}) q_{\pi}(s, a, \boldsymbol{\theta}) \frac{\frac{\partial \pi(a | s, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}}{\pi(a | s, \boldsymbol{\theta})} \\
&= \sum_{s \in \mathcal{S}} \mu_{\pi}(s, \boldsymbol{\theta}) \sum_{a \in \mathcal{A}} \pi(a | s, \boldsymbol{\theta}) q_{\pi}(s, a, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \ln \pi(a | s, \boldsymbol{\theta}) \\
&= \sum_{s \in \mathcal{S}} \mu_{\pi}(s, \boldsymbol{\theta}) \mathbb{E}_{a \sim \pi} \left[q_{\pi}(s, a, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \ln \pi(a | s, \boldsymbol{\theta}) \right] \\
&= \mathbb{E}_{s \sim \mu_{\pi_{\boldsymbol{\theta}}}, a \sim \pi_{\boldsymbol{\theta}}} \left[q_{\pi}(s, a, \boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \ln \pi(a | s, \boldsymbol{\theta}) \right]
\end{aligned} \tag{A.5}
$$

Appendix B

Demonstration of Results

For the demonstration of CoDeb's performance, this section is dedicated to presenting outputs of the system. Since the code files that undergo Compiler Checking step infers that they have passed previous checks, we will not present results of those previous steps.

B.1 Compilable Generated Code Files

Compilable results after Initial Code Generation

Figure B.1 shows a successfully compiled Java implementation of instruction CMOVNGE in the first loop of Initial Code Generation. Meanwhile, there are cases where the system produce a fraud compilable file - that is the operation part is all comment lines, e.g: the sample in Figure B.2a. However, for the case of instruction NOP (No Operation), it is, in turn, correct (Figure B.2b).

Compilable results after Iterative Code Generation

Taken from experiment Default-0, the implementation for instruction STD becomes compilable after the first loop of Iterative Code Generation. Below is the content of the prompt used to ask the Code Debugger to fix the error after the Initial Code Generation:

```
<s>[INST] <<SYS>>
You are an expert Java code debugger.
Your job is to suggest solutions to fix an erroneous program.
The program that needs correction and its error are in the input.
The suggestion is in the context.
Note that class name is changeable while file name cannot be changed.
<</SYS>>

### Input:
- The errorneous program std.java:
package v2.org.analysis.transition_rule.x86instruction;

import v2.org.analysis.environment.memory.MemoryV2;
import v2.org.analysis.environment.stack.StackV2;
import v2.org.analysis.path.BPState;
import v2.org.analysis.transition_rule.stub.X86InstructionStub;
import java.util.List;
import org.jakstab.asm.Operand;
import org.jakstab.asm.x86.X86Instruction;
import v2.org.analysis.environment.Environment;
import v2.org.analysis.path.BPPath;
import v2.org.analysis.transition_rule.X86TransitionRule;
```

```java
package v2.org.analysis.transition_rule.x86instruction;

import v2.org.analysis.environment.memory.MemoryV2;
import v2.org.analysis.environment.stack.StackV2;
import v2.org.analysis.path.BPState;
import v2.org.analysis.transition_rule.stub.X86InstructionStub;
import java.util.List;
import org.jakstab.asm.Operand;
import org.jakstab.asm.x86.X86Instruction;
import v2.org.analysis.environment.Environment;
import v2.org.analysis.path.BPPath;
import v2.org.analysis.transition_rule.X86TransitionRule;
import v2.org.analysis.value.LongValue;
import v2.org.analysis.value.BooleanValue;

public class cmovnge extends X86InstructionStub {
    @Override
    public BPState execute() {
        // From Top-most base class
        String groupName = this.groupName;
        X86Instruction inst = this.inst;
        BPPath path = this.path;
        List<BPPath> pathList = this.pathList;
        X86TransitionRule rule = this.rule;
        BPState curState = this.curState;
        Operand dest = this.dest;
        Operand src = this.src;
        Environment env = this.env;
        int opSize = this.opSize;
        List<Long> params = this.params;

        // Example syntax for retrieving register values, use them for other registers.
        LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
        LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");

        // Example syntax for retrieving flag values, use them for other flags.
        BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
        BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();

        // Retrieve stack value
        StackV2 stack = (StackV2) env.getStack();

        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // Generate from here
        LongValue srcValue = null;
        LongValue destValue = null;

        if (dest.isMemoryOperand()) {
            srcValue = memory.getValue(src.getMemoryOperand());
            destValue = memory.getValue(dest.getMemoryOperand());
        } else if (dest.isRegisterOperand()) {
            srcValue = eax;
            destValue = env.getRegister().getRegisterValue(dest.getRegisterOperand());
        }

        if (srcValue != null && destValue != null) {
            if (!CFlag.getValue()) {
                destValue.setValue(srcValue.getValue());
            }
        }

        return null;
    }
}

```

Figure B.1: Successfully compiled Java implementation of instruction CMOVNGE in the first loop of Initial Code Generation of the experiment Default-0. *Yellow box indicates the part that are not in the given template that the system generate.*

```java
public class daa extends X86InstructionStub {
    public BPState execute() {
        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // DAA code goes here
        // TODO: Implement the DAA instruction
        return null;
    }
}
```

(a) DAA instruction.

```java
public class nop extends X86InstructionStub {
    public BPState execute() {
        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // Generate from here
        // NOP instruction does not impact machine context,
        except for the EIP register.
        // Thus, the EIP register is not affected by this
        instruction.
        return null;
    }
}
```

(b) NOP instruction.

Figure B.2: Compilable Java implementation for instruction NOP and DAA in the first loop of Initial Code Generation. *NOP is correct while DAA is fraud by leaving a to-do comment only.*

```
import v2.org.analysis.value.LongValue;
import v2.org.analysis.value.BooleanValue;

public class std extends X86InstructionStub {
    @Override
    public BPState execute() {
        // From Top-most base class
        String groupName = this.groupName;
        X86Instruction inst = this.inst;
        BPPath path = this.path;
        List<BPPath> pathList = this.pathList;
        X86TransitionRule rule = this.rule;
        BPState curState = this.curState;
        Operand dest = this.dest;
        Operand src = this.src;
        Environment env = this.env;
        int opSize = this.opSize;
        List<Long> params = this.params;

        // Example syntax for retrieving register values, use them for other registers.
        LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
        LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");

        // Example syntax for retrieving flag values, use them for other flags.
        BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
        BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();

        // Retrieve stack value
        StackV2 stack = (StackV2) env.getStack();

        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // Generate from here
        CFlag.setBooleanValue(true);
        env.getRegister().setRegisterValue("eflags", CFlag);

        return null;
    }
}


- The error message:
std.java:47: error: cannot find symbol
        CFlag.setBooleanValue(true);
              ^
  symbol:   method setBooleanValue(boolean)
  location: variable CFlag of type BooleanValue
1 error


### Context:
Suggest using these syntaxes:
env.getFlag().setOFlag(new BooleanValue(true));
```

```
77
78 ### Response:
79 [/INST]
```

- Lines `#11-62` is the content of the Java code file that results from the Initial Code Generation stage. Note that this instruction's operation is short, so the generated part that the Code Writer adds in is those in lines `#57, 58`.
- The project-level context known by human developers is that there exists no method named `setBooleanValue(bool)` in class BooleanValue, and there is no universal register named `eflags` in BE-PUM.
- As the first error is to be corrected first, lines `#66-71` includes its raw message from the compiler Javac.
- To suggest the potentially related code snippet from BE-PUM, the whole erroneous code line `CFlag.setBooleanValue(true);` is used as the query into BEPUM-KB.
- The retrieval from BEPUM-KB for such query is `env.getFlag().setOFlag(new BooleanValue(true));`, which is put into line `#76` of the prompt.

Receiving the prompt, the Code Debugger replies:

```
1  The error message suggests that the variable "CFlag" cannot be found in the current scope.
       To fix this error, you can replace the line with the following:
2
3  ```
4  env.getFlag().setCFlag(new BooleanValue(true));
5  ```
6
7  This sets the CFlag value to true in the current environment.
8
9  Alternatively, you can also use the following syntax:
10
11 ```
12 CFlag.setBooleanValue(true);
13 ```
14
15 This sets the CFlag value to true, but it is not recommended to use this syntax as it is
       not clear what the CFlag variable
```

This response from Code Debugger is put into the prompt typed Iterative Code Generation of the Code Writer. The Code Writer then responses with its correction of the given erroneous file where its attempt to fix the error is the lines `#48-50`:

```
1  // std.java
2  package v2.org.analysis.transition_rule.x86instruction;
3
4  import v2.org.analysis.environment.memory.MemoryV2;
5  import v2.org.analysis.environment.stack.StackV2;
6  import v2.org.analysis.path.BPState;
7  import v2.org.analysis.transition_rule.stub.X86InstructionStub;
8  import java.util.List;
9  import org.jakstab.asm.Operand;
10 import org.jakstab.asm.x86.X86Instruction;
11 import v2.org.analysis.environment.Environment;
12 import v2.org.analysis.path.BPPath;
13 import v2.org.analysis.transition_rule.X86TransitionRule;
14 import v2.org.analysis.value.LongValue;
15 import v2.org.analysis.value.BooleanValue;
16
17 public class std extends X86InstructionStub {
18     @Override
19     public BPState execute() {
20         // From Top-most base class
```

```
21          String groupName = this.groupName;
22          X86Instruction inst = this.inst;
23          BPPath path = this.path;
24          List<BPPath> pathList = this.pathList;
25          X86TransitionRule rule = this.rule;
26          BPState curState = this.curState;
27          Operand dest = this.dest;
28          Operand src = this.src;
29          Environment env = this.env;
30          int opSize = this.opSize;
31          List<Long> params = this.params;
32
33          // Example syntax for retrieving register values, use them for other registers.
34          LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
35          LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");
36
37          // Example syntax for retrieving flag values, use them for other flags.
38          BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
39          BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();
40
41          // Retrieve stack value
42          StackV2 stack = (StackV2) env.getStack();
43
44          // Retrieve memory value
45          MemoryV2 memory = env.getMemory();
46
47          // Generate from here
48          env.getFlag().setCFlag(new BooleanValue(true));
49          // or
50          // CFlag.setBooleanValue(true);
51
52          return null;
53      }
54  }
```

We can see that the suggestion retrieved from BEPUM-KB influences the correction guidance made by the Code Debugger and the correction executed by the Code Writer. In this case, the system manages to correct the code which then passes compiler verification. However, considering the semantic side of this instance, in instruction STD, DF flag in the EFLAGS register should be the one to be set to `true`, not the CF flag. Hence, as also stated in chapter 7, semantic verification is crucial.

B.2 A Failed Case of Iterative Code Generation

We present a case of Unchanging code correction via the example of SAHF (Store AH Into Flags) instruction. The below example illustrated the input prompt passed into the Code Writer used in Iterative Code Generation stage, which is informative enough to observe both the previously compile-failed generated code (lines `#13-75`), the guidance to fix the error from the Code Debugger (lines `#80-86`), and the potentially related code snippet drawn from BEPUM-KB (lines `#90`).

```
1   <s>[INST] <<SYS>>
2   You are an expert Java programmer.
3   Your job is to fix an erroneous program given a guidance.
4   The program that needs correction and the guidance are in the input.
5   Use suggestion in the context.
6   Output the corected version of the given program only.
7   Do not explain anything before or after the program.
8   Your answer should start with ‘‘‘java mark and end with ‘‘‘ mark.
9   Provide comment for each block of codes.
10  <</SYS>>
11
12  ### Input:
13  - The errorneous program sahf.java:
14  package v2.org.analysis.transition_rule.x86instruction;
15
```

```
import v2.org.analysis.environment.memory.MemoryV2;
import v2.org.analysis.environment.stack.StackV2;
import v2.org.analysis.path.BPState;
import v2.org.analysis.transition_rule.stub.X86InstructionStub;
import java.util.List;
import org.jakstab.asm.Operand;
import org.jakstab.asm.x86.X86Instruction;
import v2.org.analysis.environment.Environment;
import v2.org.analysis.path.BPPath;
import v2.org.analysis.transition_rule.X86TransitionRule;
import v2.org.analysis.value.LongValue;
import v2.org.analysis.value.BooleanValue;

public class sahf extends X86InstructionStub {
    @Override
    public BPState execute() {
        // From Top-most base class
        String groupName = this.groupName;
        X86Instruction inst = this.inst;
        BPPath path = this.path;
        List<BPPath> pathList = this.pathList;
        X86TransitionRule rule = this.rule;
        BPState curState = this.curState;
        Operand dest = this.dest;
        Operand src = this.src;
        Environment env = this.env;
        int opSize = this.opSize;
        List<Long> params = this.params;

        // Example syntax for retrieving register values, use them for other registers.
        LongValue eax = (LongValue) env.getRegister().getRegisterValue("eax");
        LongValue ax = (LongValue) env.getRegister().getRegisterValue("ax");

        // Example syntax for retrieving flag values, use them for other flags.
        BooleanValue AFlag = (BooleanValue) env.getFlag().getAFlag();
        BooleanValue CFlag = (BooleanValue) env.getFlag().getCFlag();

        // Retrieve stack value
        StackV2 stack = (StackV2) env.getStack();

        // Retrieve memory value
        MemoryV2 memory = env.getMemory();

        // Generate from here
        AFlag.setValue((ax.getValue() & 0x40) != 0);
        CFlag.setValue((ax.getValue() & 0x100) != 0);
        BooleanValue DFlag = (BooleanValue) env.getFlag().getDFlag();
        DFlag.setValue((ax.getValue() & 0x20) != 0);
        BooleanValue OFlag = (BooleanValue) env.getFlag().getOFlag();
        OFlag.setValue((ax.getValue() & 0x80) != 0);
        BooleanValue RFlag = (BooleanValue) env.getFlag().getRFlag();
        RFlag.setValue((ax.getValue() & 0x2) != 0);
        BooleanValue SFlag = (BooleanValue) env.getFlag().getSFlag();
        SFlag.setValue((ax.getValue() & 0x8) != 0);
        BooleanValue ZFlag = (BooleanValue) env.getFlag().getZFlag();
        ZFlag.setValue((ax.getValue() & 0x1) != 0);

        return null;
    }
}


- The guidance to fix the program:

The error message indicates that the ‘getRFlag()‘ method is not available in the ‘Flag‘
    class, and it is not clear what the method is supposed to do. It is possible that the
    ‘getRFlag()‘ method has been renamed to ‘getRFlag‘ or ‘getRFlag()‘ is not implemented
    at all.

To fix the error, you can replace the line with the following code:
‘‘‘
BooleanValue RFlag = (BooleanValue) env.getFlag().getRFlag();
‘‘‘
This code retrieves the value of the ‘RFlag‘ flag from the ‘Flag‘ class

```

```
### Context:
Suggest using these syntaxes:
boolean z = ((BooleanValue) env.getFlag().getCFlag()).getValue();

### Response:
[/INST]
```

The compilation error states that method `getRFlag()` does not exist in `Flag` class by the current code base of BE-PUM. This is the case because RF flag is a system flag (Resume Flag) that should not be altered by the application side. In addition, instruction SAHF does not involve modifying the RF flag. Therefore, a correct fix for this must be to remove the two code lines related to `RFlag`. However, as can be seen from the correction guidance of the Code Debugger, the correct solution is not offered. Indeed, the next generation of Code Writer produces the exact same faulty code program.

The situation persists for another iteration and concludes with the final iteration where the Code Writer produces a response that is static-syntactically incorrect:

```
```java
BooleanValue RFlag = (BooleanValue) env.getFlag().getCFlag();
```
This code is correct because the `CFlag` field in the `Flag` class is already of type `BooleanValue`, so there is no need to typecast it again.

Alternatively, you can also use the `getValue()` method to retrieve the boolean value from the `CFlag` field, like this:
```
boolean z = ((BooleanValue) env.getFlag().getCFlag()).getValue();
```
This is a more concise way to achieve the same result.

Both of these syntaxes are valid and will work correctly. However, it is generally a good practice to use the most specific type that is appropriate for the variable, in this case, `BooleanValue`. This can help avoid any potential casting issues and make the code more readable.

I hope this helps! Let me know if you have any questions or need further assistance.

Bye!
```
```
```

Bibliography

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, mar 2008. [Online]. Available: https://doi.org/10.1145/2089125.2089126

[2] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern era—a state of the art survey," *ACM Comput. Surv.*, vol. 52, no. 5, sep 2019. [Online]. Available: https://doi.org/10.1145/3329786

[3] N. Minh Hai, M. Ogawa, and T. Quan, "Obfuscation code localization based on cfg generation of malware," in *Foundations and Practice of Security*, vol. 9482, 02 2016, pp. 229–247. [Online]. Available: https://api.semanticscholar.org/CorpusID:33555226

[4] H. Nguyen, "Automatic extraction of x86 formal semantics from its natural language description," *Master's thesis, Japan Advanced Institute of Science and Technology (JAIST) - Information Science*, 2018.

[5] A. V. Vu and M. Ogawa, "Formal Semantics Extraction from Natural Language Specifications for ARM," in *Formal Methods – The Next 30 Years*, ser. Lecture Notes in Computer Science, M. H. ter Beek, A. McIver, and J. N. Oliveira, Eds. Cham: Springer International Publishing, 2019, pp. 465–483.

[6] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," 2024. [Online]. Available: https://arxiv.org/abs/2311.10372

[7] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," Dec. 2017, arXiv:1712.05877 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1712.05877

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," Oct. 2021, arXiv:2106.09685 [cs]. [Online]. Available: http://arxiv.org/abs/2106.09685

[9] C. Garbacea and Q. Mei, "Why is constrained neural language generation particularly challenging?" 2022. [Online]. Available: https://arxiv.org/abs/2206.05395

[10] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow," May 2018, arXiv:1805.08949 [cs]. [Online]. Available: http://arxiv.org/abs/1805.08949

[11] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," Mar. 2021, arXiv:2102.04664 [cs] version: 2. [Online]. Available: http://arxiv.org/abs/2102.04664

[12] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring Coding Challenge Competence With APPS," Nov. 2021, arXiv:2105.09938 [cs]. [Online]. Available: http://arxiv.org/abs/2105.09938

[13] Z. Bi, Y. Wan, Z. Wang, H. Zhang, B. Guan, F. Lu, Z. Zhang, Y. Sui, H. Jin, and X. Shi, "Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback," Jun. 2024, arXiv:2403.16792 [cs]. [Online]. Available: http://arxiv.org/abs/2403.16792

[14] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," 2022. [Online]. Available: https://arxiv.org/abs/2207.01780

[15] I. Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, Intel Corporation, 2023, figure 2-5. System Flags in the EFLAGS Register. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf

[16] ——, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L*, Intel Corporation, 2023, page 124. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[17] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps, "Directed proof generation for machine code," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 288–305.

[18] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 829–844.

[19] N. M. Hai, M. Ogawa, and Q. T. Tho, "Packer identification based on metadata signature," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop*, ser. SSPREW-7. New York, NY, USA: Association for Computing Machinery, Dec. 2017, pp. 1–11. [Online]. Available: https://doi.org/10.1145/3151137.3160687

[20] "OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1 | BibSonomy." [Online]. Available: https://www.bibsonomy.org/bibtex/2318ba81f4fb196c34b21d95b32e2d8ae/porta

[21] T. H. M. Le, H. Chen, and M. A. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges," *ACM*

Computing Surveys, vol. 53, no. 3, pp. 62:1–62:38, Jun. 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3383458

[22] C. M. M. M.A.K. Halliday, *Introduction to Functional Grammar*. Routledge, 2013, ch. 1, 7, pp. 6, 371–372, https://doi.org/10.4324/9780203431269.

[23] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948. [Online]. Available: https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[24] D. and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. USA: Prentice Hall PTR, 2000.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[27] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *CoRR*, vol. abs/1409.1259, 2014. [Online]. Available: http://arxiv.org/abs/1409.1259

[28] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019, arXiv:1810.04805 [cs]. [Online]. Available: http://arxiv.org/abs/1810.04805

[29] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training."

[30] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[31] W. L. Taylor, ""cloze procedure": A new tool for measuring readability," *Journalism Quarterly*, vol. 30, no. 4, pp. 415–433, 1953. [Online]. Available: https://doi.org/10.1177/107769905303000401

[32] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[33] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Sep. 2020, arXiv:2002.08155 [cs]. [Online]. Available: http://arxiv.org/abs/2002.08155

[34] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[35] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner,

S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[36] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code," 2024. [Online]. Available: https://arxiv.org/abs/2308.12950

[37] J. Liu, M. Yang, Y. Yu, H. Xu, K. Li, and X. Zhou, "Large language models in bioinformatics: applications and perspectives," 2024. [Online]. Available: https://arxiv.org/abs/2401.04155

[38] J. Li, T. Tang, W. X. Zhao, J.-Y. Nie, and J.-R. Wen, "Pre-trained language models for text generation: A survey," *ACM Comput. Surv.*, vol. 56, no. 9, apr 2024. [Online]. Available: https://doi.org/10.1145/3649449

[39] S. Chen, S. Wong, L. Chen, and Y. Tian, "Extending context window of large language models via positional interpolation," 2023. [Online]. Available: https://arxiv.org/abs/2306.15595

[40] Y. Gao, C. Herold, Z. Yang, and H. Ney, "Is encoder-decoder redundant for neural machine translation?" in *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Y. He, H. Ji, S. Li, Y. Liu, and C.-H. Chang, Eds. Online only: Association for Computational Linguistics, Nov. 2022, pp. 562–574. [Online]. Available: https://aclanthology.org/2022.aacl-main.43

[41] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Guiding llms the right way: Fast, non-invasive constrained generation," 2024. [Online]. Available: https://arxiv.org/abs/2403.06988

[42] X. Chen and X. Wan, "Evaluating, understanding, and improving constrained text generation for large language models," 2024. [Online]. Available: https://arxiv.org/abs/2310.16343

[43] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2020. [Online]. Available: https://arxiv.org/abs/1904.09751

[44] I. Kulikov, A. H. Miller, K. Cho, and J. Weston, "Importance of search and evaluation strategies in neural dialogue modeling," 2019. [Online]. Available: https://arxiv.org/abs/1811.00907

[45] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," 2017. [Online]. Available: https://arxiv.org/abs/1704.04368

[46] C. Meister, T. Pimentel, G. Wiher, and R. Cotterell, "Locally typical sampling," 2023. [Online]. Available: https://arxiv.org/abs/2202.00666

[47] M. Freitag, B. Ghorbani, and P. Fernandes, "Epsilon sampling rocks: Investigating sampling strategies for minimum bayes risk decoding for machine translation," 2023. [Online]. Available: https://arxiv.org/abs/2305.09860

[48] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021. [Online]. Available: https://arxiv.org/abs/2106.08295

[49] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-Efficient Transfer Learning for NLP," Jun. 2019, arXiv:1902.00751 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1902.00751

[50] E. B. Zaken, S. Ravfogel, and Y. Goldberg, "BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models," Sep. 2022, arXiv:2106.10199 [cs]. [Online]. Available: http://arxiv.org/abs/2106.10199

[51] D. Przewlocka-Rus, S. S. Sarwar, H. E. Sumbul, Y. Li, and B. D. Salvo, "Power-of-two quantization for low bitwidth and hardware compliant neural networks," 2022. [Online]. Available: https://arxiv.org/abs/2203.05025

[52] F. Meng, Z. Wang, and M. Zhang, "Pissa: Principal singular values and singular vectors adaptation of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2404.02948

[53] Y. Li, Y. Yu, C. Liang, P. He, N. Karampatziakis, W. Chen, and T. Zhao, "Loftq: Lora-fine-tuning-aware quantization for large language models," 2023. [Online]. Available: https://arxiv.org/abs/2310.08659

[54] A. Aghajanyan, S. Gupta, and L. Zettlemoyer, "Intrinsic dimensionality explains the effectiveness of language model fine-tuning," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 7319–7328. [Online]. Available: https://aclanthology.org/2021.acl-long.568

[55] Y. Li, "Deep Reinforcement Learning," Oct. 2018, arXiv:1810.06339 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1810.06339

[56] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of Machine Learning*, 2nd ed. MIT Press, 2018.

[57] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," 2022. [Online]. Available: https://arxiv.org/abs/2203.02155

[58] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018, ch. 3, pp. 61–69, in progress.

[59] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *CoRR*, vol. abs/1602.01783, 2016. [Online]. Available: http://arxiv.org/abs/1602.01783

[60] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," *CoRR*, vol. abs/1502.05477, 2015. [Online]. Available: http://arxiv.org/abs/1502.05477

[61] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," Aug. 2017, arXiv:1707.06347 [cs]. [Online]. Available: http://arxiv.org/abs/1707.06347

[62] K. Cobbe, J. Hilton, O. Klimov, and J. Schulman, "Phasic policy gradient," *CoRR*, vol. abs/2009.04416, 2020. [Online]. Available: https://arxiv.org/abs/2009.04416

[63] C. C. Hsu, C. Mendler-Dünner, and M. Hardt, "Revisiting design choices in proximal policy optimization," *CoRR*, vol. abs/2009.10897, 2020. [Online]. Available: https://arxiv.org/abs/2009.10897

[64] P. Jana, P. Jha, H. Ju, G. Kishore, A. Mahajan, and V. Ganesh, "Cotran: An llm-based code translator using reinforcement learning with feedback from compiler and symbolic execution," 2024. [Online]. Available: https://arxiv.org/abs/2306.06755

[65] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[66] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," Sep. 2020, arXiv:2009.10297 [cs]. [Online]. Available: http://arxiv.org/abs/2009.10297

[67] M. Moskal, M. Musuvathi, and E. Kıcıman, "AI Controller Interface," https://github.com/microsoft/aici/, 2024.

[68] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017. [Online]. Available: https://arxiv.org/abs/1710.05941

[69] N. Shazeer, "Glu variants improve transformer," 2020. [Online]. Available: https://arxiv.org/abs/2002.05202

[70] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903