ErrgoEngine: A Contextualized Programming Error Analysis Translation Engine

Ned Isaiah Palacios
University of the Immaculate Conception
Davao City, Philippines
npalacios_200000000446@uic.edu.ph

James Bryan Parcasio
University of the Immaculate Conception
Davao City, Philippines
jparcasio_200000000245@uic.edu.ph

Joseph Anthony Pornillos
University of the Immaculate Conception
Davao City, Philippines
jpornillos_200000001046@uic.edu.ph

Rogelio Badiang Jr.
University of the Immaculate Conception
Davao City, Philippines
rbadiang@uic.edu.ph

Aurora Cristina Manseras
University of the Immaculate Conception
Davao City, Philippines
amanseras@uic.edu.ph

ABSTRACT

A novel approach called ErrgoEngine is presented to enhance programming error messages for novice programmers. Current solutions have limitations in providing personalized, interactive, and domain-specific assistance to address the challenges novice programmers face in understanding error messages. ErrgoEngine aims to provide a rich explanation to novice programmers, alleviating the burden of getting the right answer to their programming error with minimal intervention. Designed to be language-agnostic and leverages innovations in program analysis and language tooling, as well as relevant programming learning theories, ErrgoEngine uses regular expressions to match the input error message to a corresponding error template. It extracts relevant data and generates an error explanation and bug fix suggestions based on the template and context. Evaluation showed ErrgoEngine successfully identified and translated 42 programming errors with high test coverage. Although weaknesses were identified in semantic analysis and output generation, ErrgoEngine has the potential to simplify learning for novice programmers of all skill levels. Ongoing work aims to address these weaknesses and improve the tool's effectiveness.

CCS CONCEPTS

• **Applied computing** → **E-learning**; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

contextualized programming error analysis translation, debugging, programming error, program repair

ACM Reference Format:
Ned Isaiah Palacios, James Bryan Parcasio, Joseph Anthony Pornillos, Rogelio Badiang Jr., and Aurora Cristina Manseras. 2024. ErrgoEngine: A Contextualized Programming Error Analysis Translation Engine. In *Proceedings of Philippine Computing Science Congress (PCSC2024).* Laguna, Philippines, 8 pages.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PCSC2024, May 2024, Laguna, Philippines
© 2024 Copyright held by the owner/author(s).

1 INTRODUCTION

Software automation is becoming a vital part of almost every major industry nowadays and programming is an essential skill for students pursuing computing-related courses and is becoming a useful skill for non-tech fields[26]. For those who take programming courses, known as novice programmers[17], the skill of programming enables them to solve problems and enhances their analytical and reasoning capabilities. However, programming is a "high-level" cognitive task[51] and learning proved to be a challenge for many of the novices. One area of difficulty for novices when it comes to programming is in addressing programming errors[24, 56, 65] which requires the development of debugging skills.

Debugging is the process of finding and resolving errors in software[35]. In this process, the programmer replicates the failure for two main purposes: fault localization and bug fixing[38]. Fault localization is about identifying the cause of the failure, which can be a tedious and time-consuming task in manual processes[62, 64]. However, it is essential for maintaining software quality because the faster the bug location is identified, the faster it can be removed[2]. Techniques for fault localization include program logging, assertions, profiling[64], and stack-trace analysis[63].

Bug fixing, on the other hand, is the process of preventing failures by modifying, adding, or deleting code[38]. There are three approaches: search-based, semantic-based, and template-based[10]. Search-based bug fixing uses search algorithms to automatically replace defective code[58] with a solution that meets specific criteria, such as passing unit tests or satisfying formal specifications[16]. Semantic-based bug fixing analyzes the program's behavior to generate patches and constraints that avoid the bug[31], using a constraint solver[67]. Template-based bug fixing uses fixing templates based on corresponding bug identification, which are extracted from historical data such as bug reports and answers from question-and-answer websites[33].

Debugging a program, including fault localization and bug fixing, is similar to the processes outlined in Brown and VanLehn's repair theory. It is based on the concept that individuals acquire procedural skills by attempting to solve problems[23]. Novice programmers apply the knowledge they have gained from their classes to resolve issues. If they are successful, they acquire a new approach to address the error. If not, they encounter an impasse. To overcome the impasse, individuals may either try again later or choose to repair their knowledge through a problem-solving process[34]. As they work to identify the source of the impasse, in this case, programming errors or bugs, they seek to develop a new solution that may lead to success or present an entirely new impasse[8].

Like programming, debugging is a difficult skill to master[41] as it requires programmers to constantly use new debugging strategies as they learn[14]. Proficiency in interpreting result codes, error, and warning messages of the compiler is essential[48]. A study found that the majority of students have only fair to satisfactory performance in program tracing, a method of fault localization[25]. This may be due to fragile knowledge[53], which is described as knowledge that students may only partially know, have difficulty harnessing, or be unable to recall[41]. Text-based programming error messages can be helpful, but they often use technical vocabulary[37] and can be notoriously cryptic, leading to confusion and discouragement[5]. Error messages may also be inaccurate, imprecise, and vary between different versions of the compiler used[40].

The students of a computer studies department at a local institution in Davao City also encountered issues with debugging. Researchers conducted an in-person evaluation survey to assess the students' current level of programming experience by analyzing and solving programming errors. Three programming errors were selected and presented to the students based on their current programming proficiency. The respondents consisted of freshman students from the CCS department with 59 of 70 (84.3%) of them identifying themselves as beginners or novices. Of the 59 respondents, 16 of them (27.12%) were able to correctly identify and produce a correct solution for three programming errors, 12 respondents (20.34%) correctly identified and solved two errors, 12 respondents (20.34%) also correctly identified and solved one error, 1 respondent (1.69%) was unable to both identify and produce a solution for the errors while the remaining 18 respondents (30.5%) have varying numbers of identified and solved programming errors. Notably, 53 of the 59 respondents (89.83%) who correctly identified and solved the first programming error were less confident in identifying and solving the remaining errors. Some students sought help from their peers, while others relied on their integrated development environments (IDEs) to do fault localization through program tracing. The results of this evaluation survey highlight the need for enhanced or improved programming error messages that are more informative and easier to understand, especially for beginner and novice programmers.

Several studies have been conducted to improve the understanding of programming error messages. One study found that enhanced forms of Java error messages reduced the overall number of errors, repeated errors, and created a positive learning experience for students[4] However, other studies have shown that enhanced programming error messages did not significantly impact student learning[13, 54, 68]. This is due to various factors, such as difficulties in typing code[68], improper reporting[54], and not paying attention[13, 54]. Another concern is the lack of easy to use and understand debugging tools that could help novice programmers debug their code more quickly and easily. Traditional debuggers, such as those found in IDEs, have a steep learning curve[20] because they assume that the user already has advanced knowledge and experience with debugging tools[21, 36]. This is also evident in the use of print statements over IDE debuggers to study the behavior of the program[6].

To address the issues around programming errors and debugging tools, several studies[15, 19, 21, 32, 49, 57] have employed various methods and techniques to improve the learning curve of novice programmers when it comes to fixing errors. An automated tool called Gauntlet was developed to catch and explain the top fifty programming errors, both syntax and semantic-based, in a pre-compiler for student cadets at the United States Military Academy[15]. Backstop is another tool for enhancing runtime error messages and debugging logical errors in Java[49]. This tool was tested on a set of students with little to no programming experience, and the results showed it was proven effective despite some students initially finding the output hard to understand. Another tool called SeeC is developed as a noviced-focused debugging tool prototype for C programmers that provides runtime error checking[21]. Their tool uses a localized format string used to generate the explanations of the error, and a C compiler named Clang for creating mappings into the program's respective source code files for rich explanation. A study by Charles and Gwilliam has developed and implemented an error explainer tool for use by non-CS students to assist them in understanding and interpreting Python error messages[19]. The tool, built as a plugin into an interactive notebook environment, collects and provides plain English explanations for various error types, helping students to identify common causes and gain a better understanding of the error messages. While the paper acknowledged the lack of quantitative improvement, the students found the tool useful.

Recently, researchers have investigated the use of large language models (LLMs) to enhance the programming error messages. One study used OpenAI's Codex programming model to explain and to provide fix suggestions for given programming errors[32]. The model was able to provide comprehensible messages 88% of the time, with an average rate of correctness at 47%, with improved rates if "prompt engineering" is employed. Another study conducted by Sobania investigated the performance of ChatGPT in automatic bug fixing[57] using a benchmark which consists of small erroneous Python programs that fit in the dialog system of ChatGPT. The results showed that ChatGPT successfully solved 19 out of 40 bugs, compared to 7 bugs fixed by standard program repair methods. However, ChatGPT required additional information for most of the bugs in order to solve them, which increased its success rate by 77.5%. Despite the positive results, the researchers noted that the mental cost of verifying the bug fixes provided by ChatGPT may outweigh its advantages.

The solutions explored in these studies indicate the possibilities of analyzing and translating programming error messages, relying on "hardcoded" predefined templates or rules that use traditional string matching techniques[19, 21]. However, these methods hinder extensibility when new types of errors arise or when users want to add custom support for an error. Additionally, the tools only support a limited range of programming error types, notably syntax and semantic errors. While large language models show promise, they do not provide accurate explanations due to the lack of contextual data[32, 57] and face limitations in understanding source codes due to missing information such as the source code of related files and the libraries used. Prompt engineering can be employed to overcome these limitations[59], but it may require multiple attempts and tricks to overcome token limitations with no guaranteed consistent output[61]. Therefore, these models fall short in offering the personalized, interactive, and domain-specific assistance necessary for novice programmers without further intervention.

Newer programming language tooling technologies and analysis strategies, such as Tree-Sitter[9] and a program-feedback graph from a self-supervised, program repair neural network[66], have emerged recently and have the potential to provide a common approach to programming analysis and translation, making it easier to deploy or integrate into common software. This would greatly benefit novice programmers as these tools are more accessible and user-friendly compared to the tools and programs built from previous studies[39].

Furthermore, current programming error feedback mechanisms lack a strong foundation in programming learning theories for novice programmers[5]. To address these challenges, incorporating experiential learning theory by Kolb can be beneficial. This theory emphasizes "learning by doing" or a learning process in which knowledge is gainelective observation, abstract conceptualization, and active experimentation. By providing novice programmers with concrete learning through real-world programming error experiences, they can engage in reflective observation and gain insights into effective debugging strategies. Through abstract conceptualization, they can derive general principles and mental models for effective bug localization and fixing. Finally, active experimentation enables programmers to apply their newly formed concepts and theories in practice, refining their debugging skills[30]. This approach can enhance the understanding of programming error messages and facilitate the learning process for novice programmers.

Given the identified gaps and opportunities, the researchers aimed to develop a contextualized programming error analysis translation engine, named ErrgoEngine, to assist new programmers in effectively overcoming commond through experience[12]. It involves a four-stage process: concrete learning, ref debugging challenges. The engine will use a language-agnostic approach to enhance error programming messages by leveraging innovations in program analysis and language tooling as well as applying relevant programming learning theories. This will provide a rich and helpful explanation to novice programmers, alleviating the burden of getting the right answer to their programming error with minimal to no intervention. Ultimately, the researchers believe that ErrgoEngine will simplify the learning process for novice programmers and programmers of all skill levels, helping them transform into skilled troubleshooters.

2 APPROACH

2.1 Conceptual Framework

Figure 1 illustrates the process of error analysis and translation. Upon receiving the input programming error message text from the compiler / interpreter, it goes through the error analysis and error translation stages to produce its desired output.

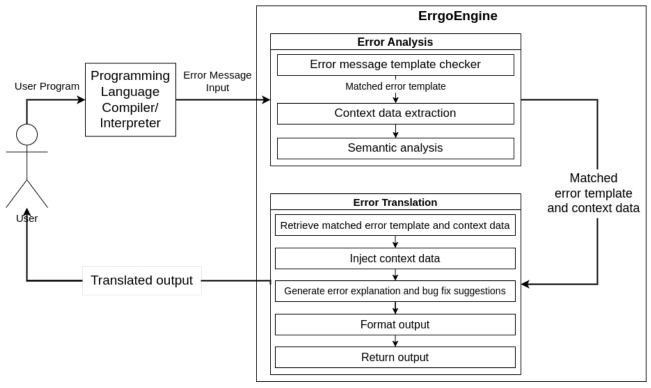

ErrgoEngine
Error Analysis
Error message template checker
Matched error template
Context data extraction
Semantic analysis
User Program
Programming Language Compiler/ Interpreter
Error Message Input
User
Matched error template and context data
Error Translation
Retrieve matched error template and context data
Inject context data
Generate error explanation and bug fix suggestions
Format output
Return output
Translated output

Figure 1: Conceptual Framework of ErrgoEngine

2.1.1 Error Analysis. In this stage, the input error message is examined and relevant data is extracted into context data. ErrgoEngine searches for a corresponding error template in its collection using a regular expression pattern included in each template. Upon discovering a match, the engine proceeds to context data extraction. If no match is found, a fallback error template is used instead. During context data extraction, variables from the input error message are obtained using the same regular expression pattern utilized in the error template matching process. If stack trace data is available, it is separately extracted using a specific regular expression pattern defined in the programming language configuration linked to the matching error template. This process extracts the locations of the offending files, whose contents are then semantically analyzed to generate a symbol table containing the program's symbols (variables, functions, classes, etc.) using the same programming language configuration used earlier. The context data, along with the matched error template, is subsequently forwarded to the error translation stage.

2.1.2 Error Translation. In the final stages, the error explanation and the bug fix for the corresponding error are generated using the error template and the context data. The error explanation is produced by executing an error explanation function, which accepts context data and returns a text output. This function can use the context data to inject values into the error explanation or selectively generate text based on the symbol information or the AST node structure of the offending code. Once the error explanation is generated, the bug fix can be produced. A function then accepts the context data and returns a list of bug fix suggestions based on the identified error through the error template upon execution. The newly-generated error explanation and bug fix suggestions are then compiled and formatted into Markdown, a plain text format for writing structured documents[1]. The formatted output is sent back to the user for consumption.

2.2 Context Data

The context data contains crucial information primarily used for the error translation stage and portions of the error analysis stage.

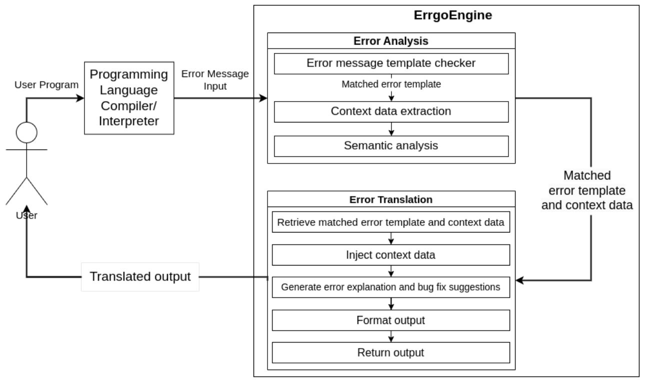
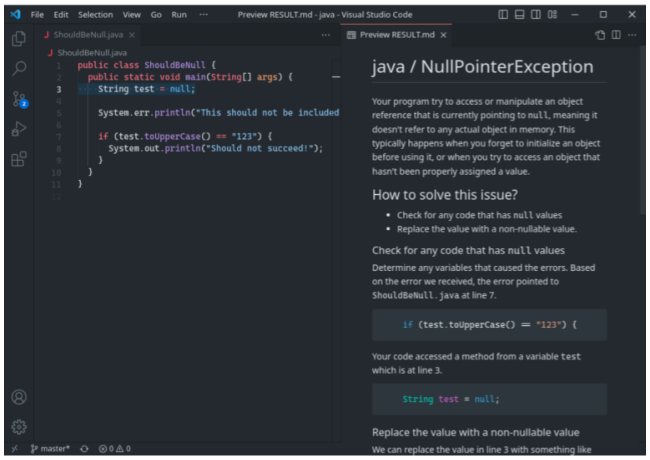

ErrgoEngine
Error Analysis
Error message template checker
Matched error template
Context data extraction
Semantic analysis
User Program
Programming Language Compiler/ Interpreter
Error Message Input
Matched error template and context data
Error Translation
Retrieve matched error template and context data
Inject context data
Generate error explanation and bug fix suggestions
Format output
Return output
User
Translated output

Figure 2: Flow diagram of the BugBuddy application


File Edit Selection View Go Run ···
Preview RESULT.md - java - Visual Studio Code
ShouldBeNull.java
Preview RESULT.md
master*

```
1  public class ShouldBeNull {
2    public static void main(String[] args) {
3      String test = null;
4
5      System.err.println("This should not be included
6
7      if (test.toUpperCase() == "123") {
8        System.out.println("Should not succeed!");
9      }
10   }
11 }
12
```

java / NullPointerException

Your program try to access or manipulate an object reference that is currently pointing to null, meaning it doesn't refer to any actual object in memory. This typically happens when you forget to initialize an object before using it, or when you try to access an object that hasn't been properly assigned a value.

How to solve this issue?

- Check for any code that has null values
- Replace the value with a non-nullable value.

Check for any code that has null values

Determine any variables that caused the errors. Based on the error we received, the error pointed to ShouldBeNull.java at line 7.

```
if (test.toUpperCase() == "123") {
```

Your code accessed a method from a variable test which is at line 3.

```
String test = null;
```

Replace the value with a non-nullable value

We can replace the value in line 3 with something like

Figure 3: BugBuddy editor example user interface

It provides guidance and personalization for the generated output. The data includes information extracted from the input error message, such as variables and stack trace locations. Additionally, it contains data extracted from semantic analysis, including files, abstract syntax trees (ASTs), and the symbol table containing the extracted symbol information.

2.3 Language-agnostic Approach

ErrgoEngine is able to identify errors and source codes written in any programming language through a supported programming language parser and an additional configuration file for that specific language. The supported programming parsers are the ones that were generated from parser generators such as ANTLR[52] and Yacc[27] as the generated parsers use common data structures that are easy to traverse through the provided mechanisms [3]. In this case, ErrgoEngine used Tree-Sitter parser as the library for parsing and programming languages with Tree-Sitter support are automatically supported.

In addition, a separate language configuration file is also created which are linked into the error templates of their respective programming errors. It contains a set of operations and variables needed in order to produce additional information helpful in identifying errors such as functions for converting stack trace positions, tracking dependency imports, and capturing symbol information as part of the semantic analysis.

2.4 Library / Application Development

ErrgoEngine is a software component library that can be used on a variety of similarly-scoped applications. To test its functionality in the real-world scenario, the researchers developed a purpose-built application called BugBuddy. Shown in Figure 2 and 3, BugBuddy is a console-based server application and editor extension that enhances programming error messages within the user's preferred text editor. It consists of multiple components:

(1) **Program Executor** - monitors incoming programming error messages via the executed program's standard error output pipe (STDERR). Error messages are sent to the daemon server for processing.
(2) **Daemon Server** - an independent, background process that acts as a main hub for receiving, processing, and dispatching output generated by ErrgoEngine. Communication between clients (e.g. program executors and language servers) is done via the JSON Remote Procedure Call (JSON-RPC) protocol[28]
(3) **Language Server** - receives and dispatches the output from the daemon server to the supported text editors and IDEs. It communicates with editors via Microsoft's Language Server Protocol (LSP)[44].
(4) **Editor Extension** - acts as a client for receiving and displaying the output from the language server to the text editor diagnostics to be displayed to the user.

The researchers chose the Go programming language[18] for developing ErrgoEngine library and BugBuddy server due to its performance, security, and package availability [50, 60]. Visual Studio Code was used for development and testing the editor extension, known for its flexibility, ease of development via its extension API[43], and compatibility with LSP[45]. TypeScript, a superset of JavaScript with type annotations[46], was used to create the extension for VSCode.

3 EVALUATION

In order to provide initial insights on the usability of ErrgoEngine, the following objectives were established:

RQ1: Test Programs and Error Templates Development. What set of test programs and error message templates or rules should be developed to thoroughly evaluate the performance of the engine?

RQ2: Contextualization Mechanism Design. How can a mechanism be designed and implemented to enable the contextualization of error messages by the translation engine?

RQ3: Real-world Use. How can the ErrgoEngine library and application be integrated with existing development tools and environments to enhance its accessibility and ease of use for programmers?

3.1 RQ1: Test Programs and Error Templates Development

3.1.1 Data Gathering. The researchers collected data from previous studies and articles to determine the programming errors when creating error templates. First, the researchers identified the programming languages that would be used in this study. Based on the current programming curriculum used by the computer studies department, the researchers chose the Java programming language. In addition to Java, the researchers also included the Python programming language as a sample implementation to test the language-agnostic capabilities of ErrgoEngine.

Afterwards, studies and articles collecting the most common programming errors of Java[4, 7, 11, 42, 47, 49] and Python[22, 29, 55, 68] programming languages from students and developers were selected, ensuring a comprehensive coverage of programming errors made from individuals with varying levels of experience. Errors that do not output an error message such as logical programming errors were excluded. Upon collection, the researchers identified 49 programming errors which consisted of 43 programming errors for Java and 6 programming errors for Python. The reason for the low number of supported programming errors for Python compared to Java is due to only it being a test implementation.

3.1.2 Test Programs. For each identified programming error, the researchers utilized ChatGPT to generate the program code that would reproduce the error. The code was then verified running it on a language interpreter. In some cases, modifications were also made to the code to ensure that the exact programming error was thrown. Actual error messages from the interpreter were also extracted for the error template design. Upon running the test programs, the researchers discovered that some errors, particularly the Java *expected* and *cannot find symbol* programming errors, produced similar messages. These errors were merged with other similar errors and kept as additional test cases for testing. As a result, the number has been reduced to 42 programming errors.

3.1.3 Error Template Design. The error template format encapsulates essential components which consists of an error name, a regular expression pattern for error message matching, and crucial functions for detecting error positions, generating explanations, and crafting bug fixes. In addition, custom regular expression patterns for error message and stack trace formats were added to handle such situations found in dynamic interpreted languages that provide different error message and stack trace formats for compile-time and runtime errors.

3.1.4 Translated Error Output Design. Shown in Figure 4 , the Markdown-based output consists of an error explanation and bug fix suggestions. Error explanation offers a brief description of the error, accompanied with the code snippet pinpointing the exact location of the error. Following this, bug fix suggestions provide one or more recommendations, each equipped with step-by-step instructions and a visualized code snippet of the fix to guide developers in resolving the identified issue.

3.1.5 Testing. The effectiveness in terms of detection and output generation was evaluated through unit testing. The test cases

~~~
# PrivateAccessError
This error occurs when you try to access a private variable from another
class, which is not allowed.
```
        // Attempting to access a private variable from another class
        int value = anotherClass.privateVariable;
                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        System.out.println(value);
    }
```
## Steps to fix
### 1. Use a public accessor method
1. To access a private variable from another class, create a public
accessor method in `AnotherClass`.
```diff
class AnotherClass {
    private int privateVariable = 10;
+
+     public int getPrivateVariable() {
+         return this.privateVariable;
+     }
+
}

```
2. Then, use this method to get the value.
```diff
        AnotherClass anotherClass = new AnotherClass();
        // Attempting to access a private variable from another class
-         int value = anotherClass.privateVariable;
+         int value = anotherClass.getPrivateVariable();
        System.out.println(value);
    }
```
This way, you respect encapsulation by using a method to access the
private variable.

### 2. Make the variable public (not recommended)
If you must access the variable directly, you can make it public, but
this is generally not recommended for maintaining encapsulation.
```diff
class AnotherClass {
-     private int privateVariable = 10;
+     public int privateVariable = 10;
}
```
~~~

Figure 4: Sample engine-translated output for Java's private access error

	Java	Python	Total
Implemented test cases	44	6	50
No. of passed tests	36 (81.81%)	5 (83.33%)	41 (73.21%)
No. of failed tests	1 (2.27%)	1 (16.67%)	2 (3.57%)
No. of skipped tests	7 (15.9%)	0	7 (15.9%)
Code coverage	92.7%	97.4%	95.05% (avg)

Table 1: Error template unit testing statistics summary

consisted of input test programs, extracted programming error messages, and the expected engine-generated output. Each identified error was given one test case. To ensure comprehensive coverage, some programming errors were also given additional test cases. These test cases were taken from the similar errors identified earlier, as well as new test cases that addressed scenarios with unexpected bugs. Based on the results shown in Table 1, 50 test cases were created in total with 44 (78.57%) test cases created for the Java programming language while 6 (10.71%) test cases were created for Python. During the test, 44 (79.55%) test cases were able to pass while there were 2 test cases (3.57%) that failed and 7 test cases

Component	Test Passed / Failed / Skipped	Code Coverage
Initial context data extraction	6/0/0	39%
Error stack trace detection	4/0/0	25.96%
Semantic Analysis - File Opening	5/0/0	37.93%
Semantic Analysis - Symbol Collection	1/0/0 - Java 1/0/0 - Python	33.3% - Java 34.2% - Python

Table 2: Contextualization mechanism implementation unit testing results

(15.9%) that were skipped due to missing standard library symbol information and lack of implementation.

3.1.6 Programming Error Coverage. The researchers were not able to fully implement all of the identified programming errors. 29 of the 36 Java programming errors (80.55%) and 5 of 6 Python programming errors (83.33%) were successfully implemented. In total, there are 34 of 42 programming errors (80.95%) that are currently implemented and supported by ErrgoEngine. Apart from the reasons mentioned earlier, the incomplete implementation support was also due to its inability to process error messages of the same programming error in different versions of the same programming language which was the case for Java. Another obstacle was in bug fixing in which the lack of code formatting information would make ErrgoEngine provide inaccurate fixes that involve adding characters or braces for example.

3.2 RQ2: Contextualization Mechanism Design

3.2.1 Implementation Testing. The researchers employed unit testing into the implementations of its contextualization mechanism which includes initial context data extraction, error stack trace extraction, and semantic analysis which is further divided into file opening and symbol collection tests. The initial context data extraction process demonstrated strengths in accurately extracting relevant information from error messages using regular expressions and passing all tests with a test coverage of 39%. However, the test coverage could be improved to ensure more comprehensive testing. The error stack trace extraction process also showed strengths in accurately extracting data from raw stack trace text and handling partially invalid inputs, but the test coverage of 25.96% was lower than that of the initial context data extraction process. The semantic analysis process revealed weaknesses in the system, including issues with feature parity of symbols collected for each supported language, limited support of the importing dependencies implementation, and inconsistencies in the order of outputs generated.

3.2.2 Error Position Adjustment using Context Data. Added at the end of the error analysis process, it is achieved by introducing an additional property into the error template. This addition is a crucial part as stated in the repair theory as the added process dictates where the error in the code occurred, utilizing information gathered during the preceding steps of error analysis. It provided enhanced fault localization especially in errors that lack column position data and improved code organization and clear separation of concerns in terms of error template implementation.

Test Case	Test Passed / Failed / Skipped	Test Coverage
Explanation generator	13/0/0	95.5%
Bug fix Generator	16/0/0	
Output Generator	8/0/0	84.3%

Table 3: ErrgoEngine error translation testing results

Component	Test Passed / Failed / Skipped	Test Coverage
Program Executor	4/2/0	93.8%
Daemon Server	15/0/0	66.1%
Information Logger	17/0/0	85.3%
Language Server	13/0/0	53.6%

Table 4: BugBuddy server unit testing results

3.3 RQ3: Real-world Use

3.3.1 Unit testing. The researchers conducted unit testing to evaluate the performance and reliability of the ErrgoEngine library and BugBuddy application. For ErrgoEngine, the researchers concentrated on testing the components involved in the error translation stage, as error analysis had already been addressed during the error template and contextual mechanism tests. The error translation stage consists of three main components: the explanation generator, bug fix generator, and output generator.

The explanation generator and bug fix generator has shown strengths in handling their respective tasks, with high test coverage rates of 95.5%. However, the output generator component showed a potential weakness, with a lower test coverage of 84.3%. While all 8 test cases were successfully passed, the lower test coverage suggests that there may be some scenarios that were not tested. Further testing may be needed to ensure the robustness of the output generator component. Overall, the high test coverage rates for the explanation generator and bug fix generator components indicate their reliability in generating informative explanations and bug fixes, but more work may be needed to improve the output generator component.

As for BugBuddy, the testing process is divided into server application tests and extension tests. For the server application tests, the researchers evaluated the core components of BugBuddy, which include the program executor, daemon server, information logger, and language server.

Results in table 4 reveal both strengths and weaknesses in these components. The program executor had a high code coverage of 93.8% but only a 66.67% pass rate due to issues with processing multiple programming errors. The Daemon Server had a perfect pass rate of 100% but a relatively low code coverage of 66.1%. The Information Logger had a high pass rate of 100% and a code coverage of 85.3%, but there were still areas of the code that were not tested. The Language Server had a pass rate of 100% but a very low code coverage of 53.6%, indicating that more testing is needed to ensure its robustness and accuracy. Overall, while the testing results demonstrate the effectiveness and reliability of the BugBuddy framework's components, there are still areas that need improvement, particularly in terms of code coverage and processing multiple programming errors. The researchers plan to address these weaknesses and continue refining the framework to improve its overall performance.

The BugBuddy extension tests, on the other hand, were all successful, including the connection from the server to the editors and receiving of translated error outputs. However, it is worth noting that the tests only validated the core functionalities of the extension, and further testing may be necessary to ensure its suitability for a wider range of projects and programming languages. Overall, the successful tests highlight the strengths of the extension as a user-friendly and robust interface for BugBuddy.

3.3.2 Initial usability test. Another test was conducted to evaluate the efficacy and user experience of the BugBuddy application. The assessment consists of three (3) student participant testers which were all given a set of sample programs that to use to test in terms of setup, functionality, and overall usability. Results indicated a generally positive reception among participants. The setup process was straightforward, with participants successfully installing and configuring both the BugBuddy application and its accompanying extension within their preferred text editors. It also demonstrated accurately in capturing and translating programming error messages during code execution, thereby assisting users in pinpointing errors and providing relevant explanations and suggestions for resolution. Participants found the explanations and suggestions to be concise, comprehensible, and beneficial for enhancing their understanding of programming concepts.

However, there are limitations of the assessment. The controlled environment and the modest sample size may not encapsulate the full spectrum of user interactions with BugBuddy in real-world programming environments. While the initial usability assessment offers valuable insights into BugBuddy's user experience, a more comprehensive performance evaluation is yet to be done right after the usability tests.

4 CONCLUSION

The evaluation of ErrgoEngine provided valuable insights into its usability and effectiveness. The researchers successfully developed a set of test programs and error templates for 42 programming errors, which were reduced from 49 after merging similar errors. The contextualization mechanism was designed and implemented, enabling the translation engine to provide more accurate and relevant error messages. The ErrgoEngine library and BugBuddy application were integrated with existing development tools and environments, enhancing their accessibility and ease of use for programmers. The unit testing results showed that the error template and contextual mechanism tests were successful, with high test coverage rates. However, there were some weaknesses identified in the semantic analysis process, which will need to be addressed in future work. The error translation stage also demonstrated strengths in handling its tasks, with high test coverage rates for the explanation generator and bug fix generator components. However, the output generator component showed potential weaknesses, and further testing may be needed to ensure its robustness. The initial evaluation results also shown it was able to capture errors and is easy to setup. Overall, the evaluation results suggest that ErrgoEngine has the potential to be a useful tool for improving the debugging process for programmers.

REFERENCES

[1] [n. d.]. CommonMark. https://commonmark.org/

[2] Rui Abreu, Peter Zoeteweij, and Arjan Gemund. 2007. *On the Accuracy of Spectrum-based Fault Localization.* https://doi.org/10.1109/TAIC.PART.2007.13 Journal Abbreviation: Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007 Pages: 98 Publication Title: Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007.

[3] Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama. [n. d.]. Reading 18: Parser Generators. https://web.mit.edu/6.005/www/fa15/classes/18-parser-generators/

[4] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education.* ACM, Memphis Tennessee USA, 126–131. https://doi.org/10.1145/2839509.2844584

[5] Brett A. Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J. Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, Janice L. Pearce, and James Prather. 2019. Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education.* ACM, Aberdeen Scotland Uk, 177–210. https://doi.org/10.1145/3344429.3372508

[6] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the dichotomy of debugging behavior among programmers. In *Proceedings of the 40th International Conference on Software Engineering.* ACM, Gothenburg Sweden, 572–583. https://doi.org/10.1145/3180155.3180175

[7] Mordechai Moti Ben-Ari. 2007. Compile and runtime errors in java. *]– : http://introcs. cs. princeton. edu/java/11 cheatsheet/errors. pdf* (2007).

[8] John Seely Brown and Kurt VanLehn. 1980. Repair Theory: A Generative Theory of Bugs in Procedural Skills. *Cognitive Science* 4, 4 (Oct. 1980), 379–426. https://doi.org/10.1207/s15516709cog0404_3

[9] Max Brunsfield. 2018. Tree-sitterIntroduction. https://tree-sitter.github.io/tree-sitter/

[10] Heling Cao, YangXia Meng, Jianshu Shi, Lei Li, Tiaoli Liao, and Chenyang Zhao. 2020. A Survey on Automatic Bug Fixing. In *2020 6th International Symposium on System and Software Reliability (ISSSR).* IEEE, Chengdu, China, 122–131. https://doi.org/10.1109/ISSSR51244.2020.00029

[11] Ioana Chan Mow. 2012. Chan Mow I (2012) Analyses of Student Programming Errors In Java Programming Courses Journal of Emerging Trends in Computing and Information Sciences 3(5) pp 740 - 749. *Analyses of Student Programming Errors In Java Programming Courses* 3 (May 2012), 740–749.

[12] Adrian Cordiner. 2021. What Is the Kolb Experiential Learning Theory? https://practera.com/what-is-the-experiential-learning-theory-of-david-kolb/

[13] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education - ITiCSE '14.* ACM Press, Uppsala, Sweden, 273–278. https://doi.org/10.1145/2591708.2591748

[14] Sue Fitzgerald, Gary Lewandowski, Renée Mccauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: Finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education* 18 (June 2008). https://doi.org/10.1080/08993400802114508

[15] T. Flowers, Curtis Carver, and J. Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. T3H/10–T3H/13 Vol. 1. https://doi.org/10.1109/FIE.2004.1408551

[16] Xiang Gao, Yannic Noller, and Abhik Roychoudhury. 2022. Program Repair. http://arxiv.org/abs/2211.12787 arXiv:2211.12787 [cs].

[17] Abdul Rahman Mohamad Gobil, Zarina Shukor, and Itaza Afiani Mohtar. 2009. Novice difficulties in selection structure. In *2009 International Conference on Electrical Engineering and Informatics.* IEEE, Bangi, Malaysia, 351–356. https://doi.org/10.1109/ICEEI.2009.5254715

[18] Google. 2011. The Go Programming Language. https://go.dev/

[19] Carl Gwilliam. 2022. error_explainer/error_explainer.py at main · carlgwilliam/error_explainer · GitHub. https://github.com/carlgwilliam/error_explainer/blob/main/error_explainer.py

[20] Mercedes Gómez-Albarrán. 2005. The Teaching and Learning of Programming: A Survey of Supporting Software Tools. *Comput. J.* 48, 2 (Jan. 2005), 130–144. https://doi.org/10.1093/comjnl/bxh080

[21] Matthew Heinsen Egan and Chris Mcdonald. 2013. Runtime Error Checking for Novice C Programmers. https://doi.org/10.5176/2251-2195_CSEIT13.03

[22] Petri Ihantola, Arto Hellas, Matthew Butler, Jürgen Börstler, Stephen Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, Miguel Rubio, Judy Sheard, Bronius Skupas, Jaime Spacco, Claudia Szabo, and Daniel Toll. 2015. Educational Data Mining and Learning Analytics in Programming: Literature Review and Case Studies. https://doi.org/10.1145/2858796.2858798

[23] InstructionalDesign.org. 2022. Repair Theory (K. VanLehn). https://www.instructionaldesign.org/theories/repair-theory/

[24] Noman Islam, Ghazala Sheikh, Ridah Fatima, and Farrukh Alvi. 2019. A Study of Difficulties of Students in Learning Programming. *Journal of Education & Social Sciences* 7 (Oct. 2019), 38–46. https://doi.org/10.20547/jess0721907203

[25] Billy Javier. 2021. Understanding their voices from within: difficulties and code comprehension of life-long novice programmers. *International Journal of Arts, Sciences and Education* 1, 1 (2021), 53–73.

[26] JobStreet.com.my. 2021. What Is Coding And Why Is It Important. https://www.jobstreet.com.my/career-resources/plan-your-career/5-reasons-why-coding-is-an-important-skill-for-your-career-growth/

[27] Stephen C Johnson. 1975. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ.

[28] JSON-RPC Working Group. [n. d.]. JSON-RPC 2.0 Specification. https://www.jsonrpc.org/specification

[29] Anne K. Kelley. 2018. *A system for classifying and clarifying Python syntax errors for educational purposes*. Thesis. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/119750 Accepted: 2018-12-18T19:48:27Z.

[30] D. A. Kolbe. 1984. Experiential learning. *New Jersey, Eaglewood Cliffs* (1984).

[31] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 163–163. https://doi.org/10.1145/3180155.3182536

[32] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. 2023. Using Large Language Models to Enhance Programming Error Messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 563–569. https://doi.org/10.1145/3545945.3569770

[33] Xuliang Liu and Hao Zhong. 2018. *Mining stackoverflow for program repair*. https://doi.org/10.1109/SANER.2018.8330202 Pages: 129.

[34] Jason M. Lodge, Gregor Kennedy, Lori Lockyer, Amael Arguel, and Mariya Pachman. 2018. Understanding Difficulties and Resulting Confusion in Learning: An Integrative Review. *Frontiers in Education* 3 (2018). https://www.frontiersin.org/articles/10.3389/feduc.2018.00049

[35] Phil Lombardi. 2023. What Is Debugging? (Plus 8 Important Strategies To Try) | Indeed.com. https://www.indeed.com/career-advice/career-development/debugging

[36] Andrew Luxton-Reilly, Emma McMillan, Elizabeth Stevenson, Ewan Tempero, and Paul Denny. 2018. Ladebug: an online tool to help novice programmers improve their debugging skills. In *Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. ACM, Larnaca Cyprus, 159–164. https://doi.org/10.1145/3197091.3197098

[37] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, Portland Oregon USA, 3–18. https://doi.org/10.1145/2048237.2048241

[38] Wes Masri. 2015. Automated Fault Localization. In *Advances in Computers*. Vol. 99. Elsevier, 103–156. https://doi.org/10.1016/bs.adcom.2015.05.001

[39] Davin McCall. 2016. *Novice Programmer Errors - Analysis and Diagnostics*. phd. University of Kent,. https://kar.kent.ac.uk/61340/

[40] Davin McCall and Michael Kölling. 2019. A New Look at Novice Programmer Errors. *ACM Transactions on Computing Education* 19, 4 (Dec. 2019), 1–30. https://doi.org/10.1145/3335814

[41] Renée Mccauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: A review of the literature from an educational perspective. *Computer Science Education* 18 (June 2008). https://doi.org/10.1080/08993400802114581

[42] Cameron McKenzie. 2022. Fix these 10 common examples of the RuntimeException in Java | TheServerSide. https://www.theserverside.com/tip/Fix-these-10-common-examples-of-the-RuntimeException-in-Java

[43] Microsoft. 2015. Why Visual Studio Code? https://code.visualstudio.com/docs/editor/whyvscode

[44] Microsoft. 2016. Official page for Language Server Protocol. https://microsoft.github.io/language-server-protocol/

[45] Microsoft. 2019. Language Server Extension Guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[46] Microsoft. 2020. Documentation - TypeScript for the New Programmer. https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html

[47] Kabiru Mijinyawa, Murtala Mohammed, Abdulwahab Lawan, and Bashir Galadanci. 2015. Detection and Categorization of Errors by Novice Programmers in a First Year Java Programming Class.

[48] I.T. Chan Mow. 2008. Issues and Difficulties in Teaching Novice Computer Programming. In *Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education*, Magued Iskander (Ed.). Springer Netherlands, Dordrecht, 199–204. https://doi.org/10.1007/978-1-4020-8739-4_36

[49] Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. [n. d.]. Backstop: A Tool for Debugging Runtime Errors. ([n. d.]).

[50] Jacek Olszak and Karolina Rusinowicz. 2022. Why Golang may be a good choice for your project. https://codilime.com/blog/why-golang/

[51] Tom Ormerod. 1990. Human Cognition and Programming. Elsevier, 63–82. https://doi.org/10.1016/B978-0-12-350772-3.50009-4 Book Title: Psychology of Programming.

[52] T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810. https://doi.org/10.1002/spe.4380250705 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380250705.

[53] David Perkins and Fay Martin. 1985. *Fragile Knowledge and Neglected Strategies in Novice Programmers. IR85-22*. Technical Report. https://eric.ed.gov/?id=ED295618 ERIC Number: ED295618.

[54] Raymond S. Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, Seattle Washington USA, 465–470. https://doi.org/10.1145/3017680.3017768

[55] David Pritchard. 2015. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM, Pittsburgh PA USA, 1–8. https://doi.org/10.1145/2846680.2846681

[56] Siti Rosminah, siti rosminah md derus, and Ahmad Zamzuri Mohamad Ali. 2012. Difficulties in learning programming: Views of students. https://doi.org/10.13140/2.1.1055.7441

[57] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. http://arxiv.org/abs/2301.08653 arXiv:2301.08653 [cs].

[58] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 727–738. https://doi.org/10.1145/2950290.2950295

[59] Team GyaniPandit. 2023. Advantages and Disadvantages of Prompt Engineering. https://gyanipandit.com/programming/advantages-and-disadvantages-of-prompt-engineering/ Section: Prompt Engineering.

[60] The Go Programming Language. 2011. Go Packages. https://pkg.go.dev/

[61] Todd Kelsey PhD. 2023. 20 Prompt Engineering Paint Points | LinkedIn. https://www.linkedin.com/pulse/20-prompt-engineering-paint-points-todd-kelsey-phd/

[62] Iris Vessey. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5 (Nov. 1985), 459–494. https://doi.org/10.1016/S0020-7373(85)80054-7

[63] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE Computer Society, USA, 181–190. https://doi.org/10.1109/ICSME.2014.40

[64] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[65] Stelios Xinogalos. 2016. Designing and deploying programming courses: Strategies, tools, difficulties and pedagogy. *Education and Information Technologies* 21, 3 (May 2016), 559–588. https://doi.org/10.1007/s10639-014-9341-9

[66] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. http://arxiv.org/abs/2005.10636 arXiv:2005.10636 [cs, stat].

[67] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (Feb. 2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[68] Zihe Zhou, Shijuan Wang, and Yizhou Qian. 2021. Learning From Errors: Exploring the Effectiveness of Enhanced Error Messages in Learning to Program. *Frontiers in Psychology* 12 (2021). https://www.frontiersin.org/articles/10.3389/fpsyg.2021.768962