ASSURING CORRECTNESS, TESTING, AND VERIFICATION OF X-COMPILER BY INTEGRATING COMMUNICATING STREAM X-MACHINE

BASHIR A. SANUSI[1], EMMANUEL K. OGUNSHILE[2], MEHMET AYDIN[3], and STEPHEN O. OLABIYISI[4]

[1,2,3]Department of Computer Science and Creative Technologies, University of the West of England, Bristol, United Kingdom.
[1]bashir.sanusi@uwe.ac.uk
[2]Emmanuel.ogunshile@uwe.ac.uk
[3]Mehmet.aydin@uwe.ac.uk
[4]Department of Computer Science, Ladoke Akintola University of Technology, Ogbomoso, Nigeria.
[4]soolabiyisi@lautech.edu.ng

Abstract: Compiler design plays an important role in ensuring that the translation of the programs written in high-level language into executable code is correct. However, in todays' safety-critical environments, security gaps, visible and hidden defects in compiler models are liability factors that needed to be addressed in the process of examining that a system meets specifications and requirements of its intended objectives. Nevertheless, defects in this model are errors in coding which bring about incorrect or unexpected results from a compiler design that does not meet the actual requirements. Hence, this paper developed a novel approach by integrating the computational power of Communicating Stream X-Machine (CSXM) to address the problem associated with compiler correctness. In addition, this paper outlines the development of a novel compiler design called X-compiler, emphasizing its important role in software development. CSXM technique was implemented in Visual Studio (2022) to develop the X-compiler. The X-compiler utilize state-of-the-art techniques to translate program source code written in high-level programming language which is the blend of C# and Python programming language into executable code. The development process includes the lexical analysis, syntax parsing, semantic analysis, and code generation ensuring the correctness of the X-compiler. The results of the compiled code were run to produce output in command Windows environment for user interactions. Also, highlights the potential for enhanced language interoperability and the development of efficient compiler that leverage the strengths of multiple programming paradigms. Overall, the study on CSXM contributed to a deeper understanding of concurrent systems, while the design and implementation of the compiler showcased the feasibility of creating a synergistic blend of C# and Python programming languages.

CCS CONCEPTS • Software and its engineering • Theory of Computation • Computing methodologies.

Additional Keywords and Phrases: *Compiler, Communicating Stream X-Machine, X-Compiler, Software Development, Code Generation*.

---

1. INTRODUCTION

Over the past decades, the compiler correctness is significant to the software engineering of the safety critical software [9]. However, compilers are crucial tools, this is simply because they are the principal piece of infrastructure for constructing or building some other software [7]. That is, nearly all program running on a computer for example, operating systems, web browsers, products developed by software programmers etc, has been executed with the help of a compiler. Notably, the compilers simply translate computer programs written in high-level language e.g., Java, C++, Python etc., into representations executable by the computer system [13].

According to [2], the compilers and operating systems establish the fundamental interfaces in between software programmer, and the machine itself. Having said that, the perception of these correlations relieves the certain transitions to new software and hardware programming languages. Also, improves the ability of a software programmers to make appropriate trade off in design specification and implementation. Software programmers are known to write codes in different programming languages such as Java, C# etc to achieve a particular task. In the process of writing the code, it might contain defects depending on its complexity. This pave way for research in past decades in the field of software testing i.e., detecting, or predicting defects in a software programs [20]. [18] focuses on predicting defect-type proneness in Java programs to model and determine several types of run time defects in Java programs using Stream X-machine (SXM).

Meanwhile, the accuracy of an implementation in any given programming language depends on the compiler that serves as interface between the programmer and machine code. Recently, some research has been done on optimizing and improving the quality of compilers but the existence of defects or bugs in compilers remains crucial [14]. These compiler defects can give rise to incorrect binary code that is generated from the correct source code [27]. An example is the story behind the Java Seven GA bugs affecting Apache Lucene or Solr [26].

In some cases, the compiler defects can be injected intentionally to breach the security of the compiled applications. An example is when Apple scrambles after forty malicious XcodeGhost apps spook App store (Dan, 2015). Meanwhile, the compiler defects cannot only give rise to undesired behaviour with respect to some critical end-result, but also make the process of software debugging more difficult. Mostly, software programmers find it difficult to check or determine if defects in a software are caused by the software program itself or because of the compiler being used [4]. Based on these examples, this research focused on the direction of detecting the compiler defects i.e., features of compiler that are likely to introduce defects such as syntax, semantic etc rather than a particular programming language run time defects.

2. STATEMENT OF PROBLEM AND MOTIVATION

Correctness, testing, and verification of compilers is an area that is generating a great deal of interest among software programmers and a resource for researchers who are interested in understanding the open challenges and the state of the art in this field which can as well be applied to other areas. It is obvious that all software development relies on compilers i.e., translating the high-level language into machine or binary code, notwithstanding defects in them might be difficult to detect which can cause silent failure [5]. An example is where a software program appears to function but has been subtly mis-compiled. However, a compiled program may function incorrectly even though the original source code looks entirely correct.

Over the years, it has been established that compilers are one of the most important software infrastructures. Whereas compiler testing is a widely used technique to assure the quality of compilers [6]. Different compiler testing methods or techniques have been in existence and proposed such as "*Identifying Compiler and Optimization Level in Binary Code from Multiple Architectures"* [19], CoTest [11], as well as [24], [3], Orion [9] for generating test programs but they are limited to C programming language etc. These methods still suffer from serious efficiency problems because the techniques need to run many randomly generated test programs on a go through automated test-generation tools to effectively detect compiler defects [1].

Although, different compilers have been written for many languages such as GNU Compiler Collection (GCC), Low-Level Virtual Machine (LLVM), Javac, some of them have been used for decades but may still be buggy [22]. Another issue is that the desired behaviour of the existing tools was only informally specified i.e., lack of formal specification on what exactly the compiler is supposed to do [10]. An example is testing whether an input program gives the actual output, check if it is the same as the expected output and defects detected during the compilation process are recorded for optimization purposes.

Some other issues with the state of art are mis-compilation bugs (this is where a program appears to work but is precisely mis-compiled), returning incorrect results to clients (this is a key challenge in compiler i.e., lack of oracle that classifies an output as correct or incorrect), invalidating verification guarantees (a compiler that is accompanied by a machine-checked proof i.e., the generated machine code behaves as specified) etc [15].

Having reviewed the current state of the art, this paper is motivated and experiments the strength of formal method i.e., Communicating Stream X-machine (CSXM). First, this paper used C-Sharp (C#) programming language to build a new compiler to implement CSXM testing procedures. In this process, this paper hereby creates a compiler called X-Compiler which is a blend of C# and Python programming language. Thus, X-Compiler have all the necessary components of a functional compiler which implements its own TokensAnalyzer, Parser, Abstract Syntax Tree (AST), testing mechanisms including CSXM testing procedures.

3. RELATED WORK

[12], conducted a systematic and comprehensive empirical comparison of three compiler testing technique i.e., Randomized Differential Testing (RDT), Different Optimization Level (DOL), and Equivalence Modulo Inputs (EMI). This study only focuses on GNU Compiler Collection (GCC) and Low-Level Virtual Machine (LLVM).[24], proposed randomized differential testing technique to detect compiler warning defects. This study is limited to C compilers. [16], present quantitative and qualitative study of the impact of mis-compilation bugs in a mature compiler. The study only targets C/C++ compilers i.e., Clang and LLVM. [7] approach called HiCOND was used to generate test programs which are likely to be erroneous. This approach detects 75.00%, 133.33%, and 145.00% more bugs than DefaultTest, OriSwarm, and VarSwarm respectively. The study was unable to find different types of bugs.

[8], presented a large-scale empirical study on open-source Java projects. The study was able to investigate compiler defect types and reported their fic time. This study is limited to manual analysis of common compiler defects. [23], describes a novel SXM verification and testing method which automatically checks the specifications for completeness and determinism, prior to generating complete test suites. It is limited to Java programming language.[25], propose A Diversity-guided PROgram Mutation (DIPROM) to construct diverse warning-sensitive programs for effective compiler warning defect detection which outperformed HiCOND, Epiphron, and Hermes by up to 18.93% ~76.74% in terms of the bug-finding capacity on average. The approach is only evaluated on C compilers i.e., GCC and Clang.

[22], presented a novel specification-based testing method called SpecTest to better utilize semantics for testing. It is only applied to Java and Solidity compiler. Researchers and software programmers failed to focus on the question "Is it the source program that contains defects or the compiler itself used in the compilation process". Our findings focus on the significance and characterise different pattern of compiler defects. To the best of our knowledge, assuring the

quality of compiler (compiler testing) field has two critical tasks namely, ability to generate test cases that are likely to trigger defects and ability to find different types of compiler bugs. Having reviewed related works, these studies only focus on different approaches in generating test programs for compiler testing and most of these approaches are language dependent. From the literatures, it is clear that the researchers only focus on existing compilers such as GCC, LLVM, Clang etc. To fill this gap, we presented an approach that focused on these two challenges. First, a formal verification method (CSXM) was used to model different types of compiler defects and generate random test cases.

4. METHODOLOGY

In the process of developing X-compiler with the integration of Communicating Stream X-Machine (CSXM) method, this involves a systematic and well-organized research approach to ensure the correctness and efficiency of a language translation system. X-compiler is a new compiler developed in this paper that integrates the features of C# programming language of .Net framework constructors and the full-fledged utilization of the Python programming language structures using CSXM testing mechanisms.

4.1 Stages of the developed x-compiler

This research approach typically involves the following stages as illustrated in figure 1 which helps to systematically address different stages or phases of the developed X-compiler. In addition, all these stages convert the source code by dividing into tokens, creating parse trees, and optimizing the source code following the highlighted stages:

Figure 1: Phases of Compiler [17].

i. Language Design and Specification: In this stage of the development of X-compiler for a new language i.e. a blend of C# and Python programming language structure, this paper design and specify the syntax and semantics of the source language. At this phase, readability and ease of implementation were considered in defining the grammar and language construct. Also, this research designed a formal language specification (CSXM) having studied and investigated the formal language from theory to practice and build an Abstract Syntax Tree (AST) for representation.

ii. Lexical Analysis and Syntax Analysis: The scanner and parser components also referred to as lexical analysis and syntax analysis respectively were developed. The lexical analysis was implemented by tokenizing the source code into meaningful units while the syntax analysis was implemented by creating a parse tree also called AST which represent the structure of the source code.

iii. Semantic Analysis: In order to enforce language rules and perform type checking, the semantic analysis was implemented which check the correctness of the source code or program with respect to the defined language specifications. At this phase, this paper was able to implement symbol table management to store information about variables, functions, and types.

iv. Intermediate Code Representation: In this phase, for a well-organized code generation, an intermediate code representation was implemented, and a module was developed to generate intermediate code from the parsed and semantically analysed program.

v. Optimization and Code Generation: In this paper, optimization technique was implemented to enhance the performance of the generated program. Here the data flow analysis and register allocation were

investigated. Also, the code generation module was implemented to translate the intermediate code into machine code (target language).

vi. Defect Handling and Debugging: In this stage, this paper implements defect handling mechanisms to provide relevant detection to users. These debugging mechanisms were used to facilitate the detection and correctness of the compilation errors. This simple means when the above stages have been properly done, the X-compiler was able to compile a source of code in ".xc" format where XC simple means (X-compiler) and convert the codes to assembly language (by an assembler, interpret the assembly codes correctly and spots errors in execution. Furthermore, this X-compiler runs the code and the output on Window for the user to interact with ease.

vii. Testing and Verification: In order to ensure the testing and verification of the X-compiler, this paper created an extensive test suite (sample project codes) covering different aspects of the language and X-compiler which was used in testing and verifying the designed compiler. Also, this research performs comprehensive testing which includes unit testing, integration testing, and verification against some standard programs.

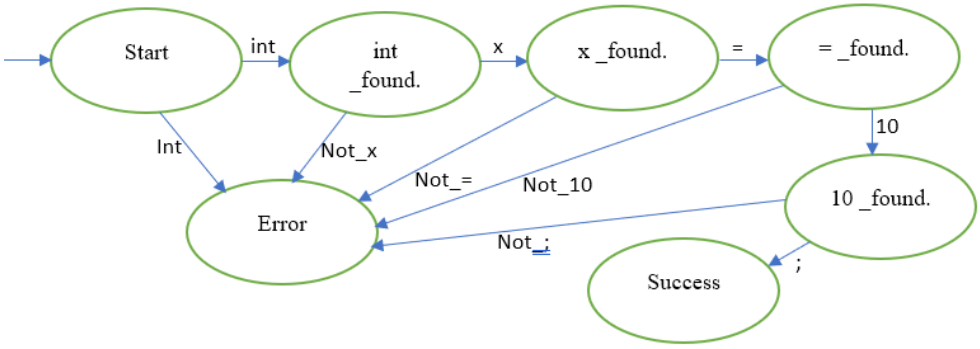
viii. Evaluation and Optimization: This paper evaluates the performance of the X-compiler using some standard programs with respect to speed and memory usage.

ix. Maintenance: This paper established and considered potential plans for ongoing maintenance (support) and modifications (extension) respectively to accommodate evolving language standards.

4.2 Modelling compiler errors using stream x-machine method

In this paper, CSXM method was used to model some compiler errors, and generate test cases that triggered errors to enhance and effectively train and test the Machine Learning model at the later stage of this research. Having studied and investigated the CSXM methodology from theory to practice, this paper presents some examples of compiler errors as presented in Figure 2. To the best of our knowledge, the formal specification (CSXM) was initialized by investigating its syntax, rules, and constraints that defines the language structure.

Example 1: Improper Casing: It's case sensitive. For example, int x = 10;



Example 2: Mismatched brackets. For example, system.out.println x);

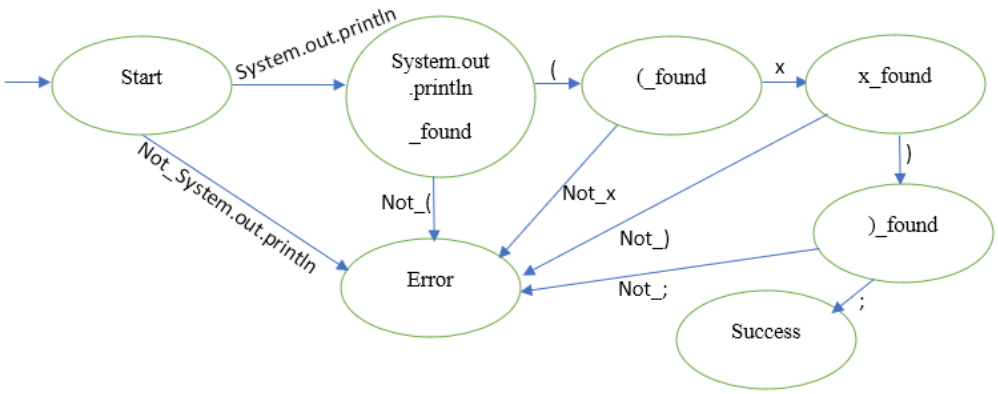


Figure 2: Some Compiler errors Modelled using CSXM technique.

4.3 Design and implementation of x-compiler

Having studied and investigated SXM and CSXM from theory to practice, as discussed in [21]. This paper was confident to apply the strength of CSXM on X-compiler. The X-compiler involve the C# static typing which enforce type safety at the compile-time. It consists of Python features like meaningful spaces, avoidance of braces, and avoidance of semicolons. It also involves no self-parameters in the declarations of all the classes and methods. It is known that a compiler is a program that represents or translates text that is entered by the user in the form of an executable file or a class module for execution on a computer. However, this translation is carried out by representing the source text in the form of an Abstract Syntax Tree (AST). In this paper, the X-compiler compilation process involves 3 steps.

Step 1: X-compiler would first translate the source text into AST, which was consumed and understandable by a computer.

Step 2: Then, it performs static checks on the source text to prevent generation of erroneous code.

Step 3: And finally, generate executable code.

AST serves as beautiful program output which helps in code restructuring, and calculation of different program metrics. For the success of the above listed operations, tree nodes that are assignment operators were treated differently than nodes that variables or arithmetic expressions. Therefore, this paper was able to create a class for each of the above entities. Below Figure 3 is the class diagram which represents the AST Node hierarchy levels.

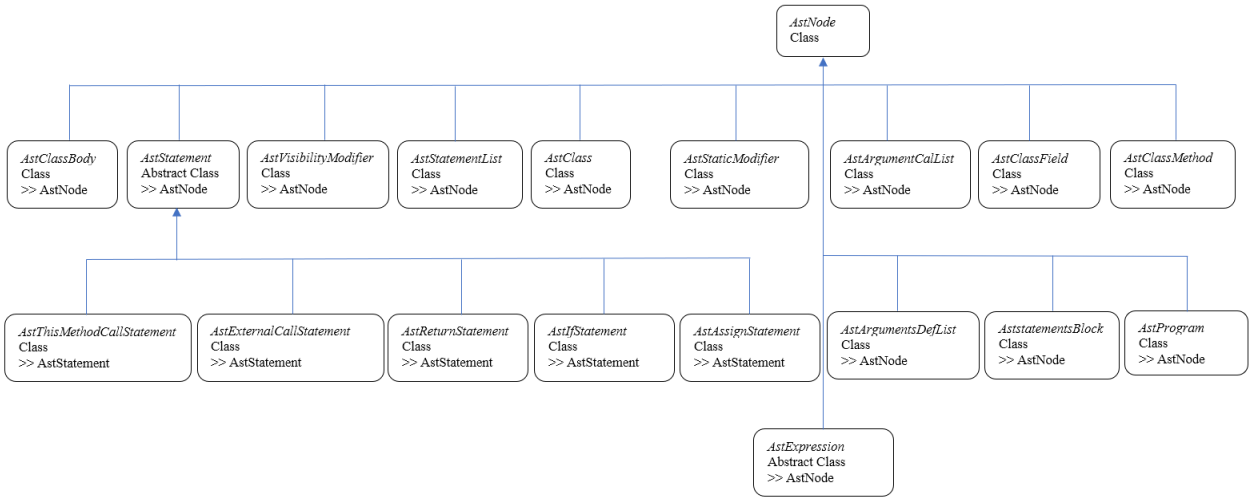


Figure 3: Class Diagram Representing the AST Node Hierarchy Levels.

The class *AstNode* is created to handle the *TextPositions* of the source files and can call it as a function without the Get and Set method. The class *AstNodeVisitor* was implemented with a stack instead of an array or vector and this was used to handle different Boolean values that were used for various validations in the program source code. The *ArrayInitializerStatement* is a Boolean for the initialization of structural arrays. Then, the class *Enums.cs* represents a class for enumerations e.g. *VisibilityModifier* which could be either public or private. *BoolValues* enumerations and *CompareOp* stands for *Operator Comparer*.

All in all, as highlighted in section A, the X-compiler methodology involves different stages in which two of these phases are the most important modules in these processes are the parser and the scanner. In this paper, the adopted methodology makes sure the program source code is transformed into a machine or an executable form while adhering to the rules and semantics of the blend of C# and Python programming language. In essence, every phase plays an important role in this transformation process as presented in figure 4.

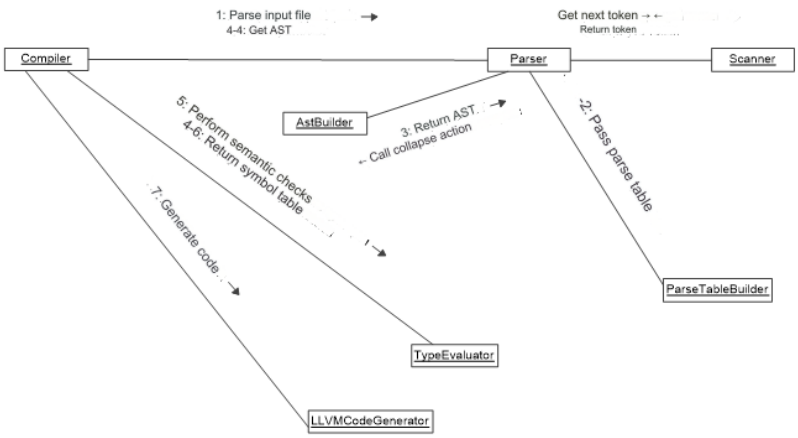


Figure 4: The Flow Diagram of the X-compiler.

5. RESULTS AND DISCUSSION

The stages involved in this paper were implemented using Visual Studio 2022 and was run on Hewlett-Packard (HP) Spectre x360 Convertible with Intel(R) Core(TM) i7-8565U, Windows 10 Home 64-bit Operating System (OS), Central Processing Unit (CPU) with a speed of 1.80GHz, 1992 Mhz, 4 Core(s), 8 Logical Processor(s), 16GB Random Access Memory (RAM), and 500 Gigabyte (GB) hard disk drive. In this paper, the design stage focused on synthesizing the distinctive features of C# and Python programming languages into a unified X-compiler structure. The design aimed to leverage the strengths of both languages, such as C#'s performance and type safety along with Python's readability and flexibility. The X-compiler design encompasses lexical analysis, syntax parsing, semantic analysis, intermediate code generation, optimization, and code generation. Careful consideration was given to achieving seamless interoperability between C# and Python constructs, ensuring a coherent and efficient compilation process.

Also, the implementation phase involved translating the designed X-compiler structure into a functional software artifact using the C# programming language. The coding process adhered to the principles and specifications outlined in this paper. This implementation included the development of classes or modules for lexical analysis, parsing, semantic processing, intermediate code generation, and the generation of the target code as stated in section IV. Comprehensive testing procedures were employed to validate the correctness and efficiency of the X-compiler. Therefore, the implementation demonstrated the successful integration of C# and Python programming language features, providing a versatile tool for upcoming and senior developers.

The Graphic User Interface (GUI) of the X-compiler is presented in figure 5. The title bar is named "X-compiler by Bashir Adewale Sanusi" while the menu bar contains the following *OPEN X-SCRIPT*, *SAVE X-SCRIPT*, *COMPILE X-SCRIPT*, *RUN X-SCRIPT*, *ABOUT*, and, *CLOSE*.

Where:

OPEN X-SCRIPT: This button is used to open a script saved to the disk and for a re-run.

SAVE X-SCRIPT: This button is used to save a script to disk for a re-run.

COMPILE X-SCRIPT: This button is used to compile a script and use the X-machine to detect erroneous codes.

RUN X-SCRIPT: This button is used to call *BttCompileXScript* to compile a script, use *XMachine* to detect errors. Then, it used LLASM-Assembler to convert the codes to assembly language and binary files. It also creates a console window to run and interact with our written program to binary files.

ABOUT: This button gives brief information about the author and the version of the X-compiler.

CLOSE: This button is used to exit the X-compiler environment.

Moreso, the task bar contains CHANGE IDE THEME and CHANGE IDE COLOR SCHEME which is used to change the Windows console theme and the change the Windows color scheme respectively.

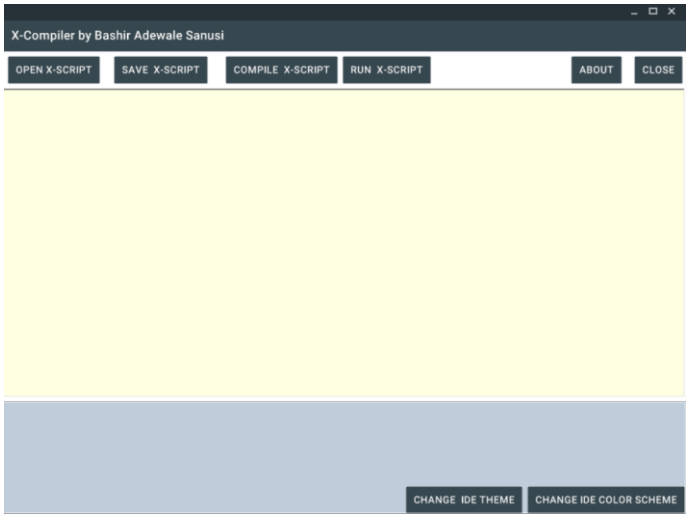


Figure 5: X-compiler Graphic User Interface.

As presented in figure 4, the flow diagram of the X-compiler involves understanding the sequential steps and these components include the processing of the program source code as illustrated in figure 6. Some sample program codes were used to test and verify the correctness of the X-compiler, where the process starts with program source code. Then, the scanner reads the program source code and generates a stream of tokens (this represents the fundamental language elements such as keywords, identifiers, and literals). These streams of tokens are passed to the parser. The parser analyzes the streams of tokens and generates an AST which executes the grammatical structure of the blend of C# and Python programming languages.

Furthermore, the semantic checker performs the semantic analysis on the AST, and it checks for the semantic errors including type mismatches and executes the language-specific rules. The AST is simply referred to as the hierarchical structure of the program source code as presented in figure 3. Additionally, the intermediate code generator translates the AST into an Intermediate Representation (IR) where the IR is a platform-independent program code representation. The code optimizer improves the efficiency of the intermediate program code, and it performs different optimizations that includes dead code elimination and constant folding. Also, the code emitter translates the optimized intermediate code into target-specific assembly or machine code and the target code is referred to as the low-level code specific to the target architecture as implemented in the *LLVMCodeGeneration.cs* class. Hence, these correctly produces an executable program as output which was run on the X-compiler.

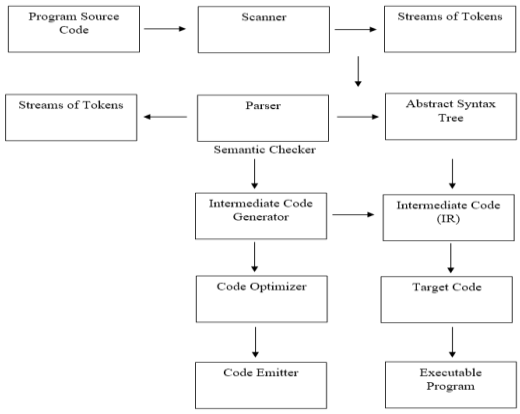


Figure 6: The Flow Diagram of the X-compiler Compilation Process.

It is obvious that compiled program source code is written in a high-level programming language and then translated into machine code. Different programming languages and compilers have been in existence. However, a novel compiler called X-compiler was designed and implemented in this paper, introducing a new blend of C# and Python programming languages which is compiled and run. A sample program code (X-Machine Calculator) is presented in algorithm 1 below which assures the correctness, testing and verification of the X-compiler. Given the sample program code, when the X-compiler compiles the program code, then translates the human-readable code into

machine-readable instructions as presented in algorithm 2 for execution. The transformed executable binary code is being run and produces the output in a console window environment as presented in figure 7.

Algorithm 1: Program Source Code for "X-Machine Calculator".

```
class xMachineCalculator:
   private int action
   private int a
   private int b

   public static int Main():
      while (action != 4):
         EnterNumbers()
         action = AskMenu()
         Console.WriteLine()
         if (action == 0):
            ProcessAdd()
         if (action == 1):
            ProcessMul()
         if (action == 2):
            ProcessDiv()
         if (action == 3):
            ProcessMod()
         Console.WriteLine()
         Console.WriteLine()
      return 0

   private int PrintLn(string msg):
      Console.WriteString(msg)
      Console.WriteLine()
      return 0

   private int AskMenu():
      PrintLn("Enter number of operations :")
      PrintLn("   0 - a + b")
      PrintLn("   1 - a * b")
      PrintLn("   2 - a div b")
      PrintLn("   3 - a mod b")
      PrintLn("   4 - exit")
      Console.WriteString("?> ")
      return Console.ReadInt()

   private int EnterNumbers():
      Console.WriteString("Enter a:> ")
      a = Console.ReadInt()
      Console.WriteString("Enter b:> ")
      b = Console.ReadInt()
      return 0
```

Algorithm 2: Machine Code for "X-Machine Calculator".

```
@.strconst0 = internal constant [29 x i8] c"Enter number of operations :\00"
@.strconst1 = internal constant [14 x i8] c"   0 - a + b\00"
@.strconst2 = internal constant [14 x i8] c"   1 - a * b\00"
@.strconst3 = internal constant [16 x i8] c"   2 - a div b\00"
@.strconst4 = internal constant [16 x i8] c"   3 - a mod b\00"
@.strconst5 = internal constant [13 x i8] c"   4 - exit\00"
@.strconst6 = internal constant [4 x i8] c"?> \00"
@.strconst7 = internal constant [11 x i8] c"Enter a:> \00"
@.strconst8 = internal constant [11 x i8] c"Enter b:> \00"
@.strconst9 = internal constant [9 x i8] c"a + b = \00"
@.strconst10 = internal constant [9 x i8] c"a * b = \00"
@.strconst11 = internal constant [11 x i8] c"a div b = \00"
@.strconst12 = internal constant [11 x i8] c"a mod b = \00"
@.spacestr = internal constant [2 x i8] c"\20\00"
@.endlstr = internal constant [2 x i8] c"\0A\00"
@.str = internal constant [3 x i8] c"%d\00"
@.rstr = internal constant [3 x i8] c"%d\00"
@.emptystr = internal constant [1 x i8] c"\00"
declare i32 @printf(i8 *, ...)
declare i32 @scanf(i8*, ...)
declare i32 @rand()
declare void @srand(i32 *)
declare i32 @time(i32 *)
declare i8* @strstr(i8*, i8*)
declare i8* @strncpy(i8*, i8*, i32)
declare i8* @memset(i8*, i8, i32)
@action = private global i32 0
@a = private global i32 0
@b = private global i32 0
define i32 @main()
br label %whilecond1
whilestart1:
%1 = call i32 @EnterNumbers()
%2 = call i32 @AskMenu()
store i32 %2, i32* @action
%3 = getelementptr [2 x i8]* @.endlstr, i64 0, i64 0
%4 = call i32 (i8 *, ...)* @printf(i8* %3)
%5 = load i32* @action
%6 = add  i32 0, 0
%7 = icmp eq i32 %5, %6
%8 = icmp eq i1 1, %7
br i1 %8, label %then1, label %else1
then1:}
```

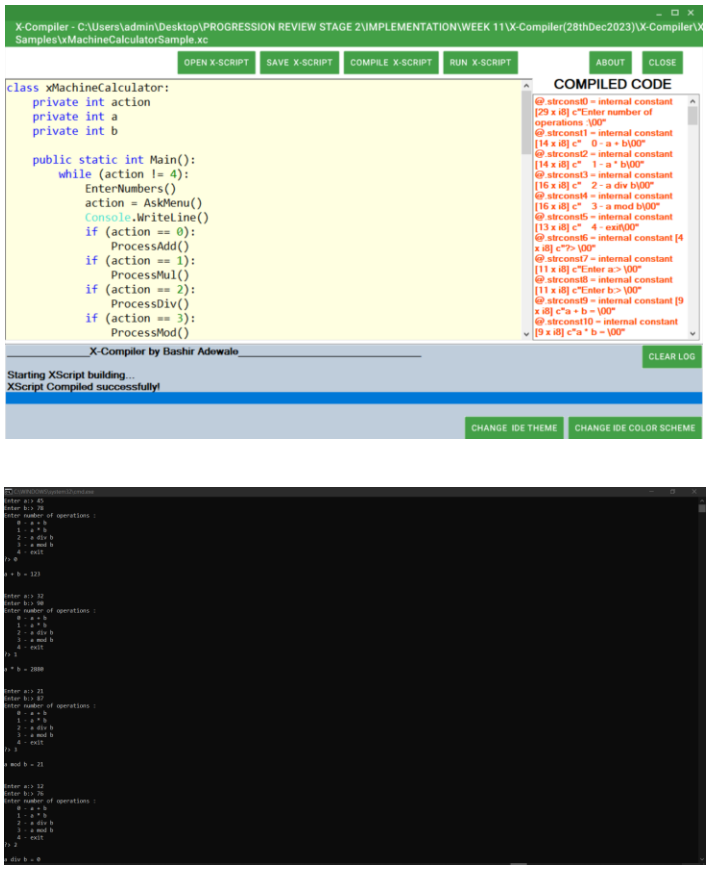


Figure 7: The Compiled Code and Console Windows GUI for "X-Machine Calculator".

Additionally, further testing of the X-compiler was carried out by copying the program source code into a C# compiler such as Visual Studio and NET Compiler Platform (Roslyn) and these codes do not compiler nor run. Also, these source codes are copied into Python compiler such as PyCharm, it does not compile nor run as well. This paper designed and implemented X-Compiler with various types of parsing mechanisms while using the static type of the blend of C# and Python programming languages. In summary, errors were injected intentionally into the sampled program source code to test the assurance and correctness of the X-compiler to see the effectiveness of the X-compiler in detecting errors during the compilation process as presented in figure 8 below.

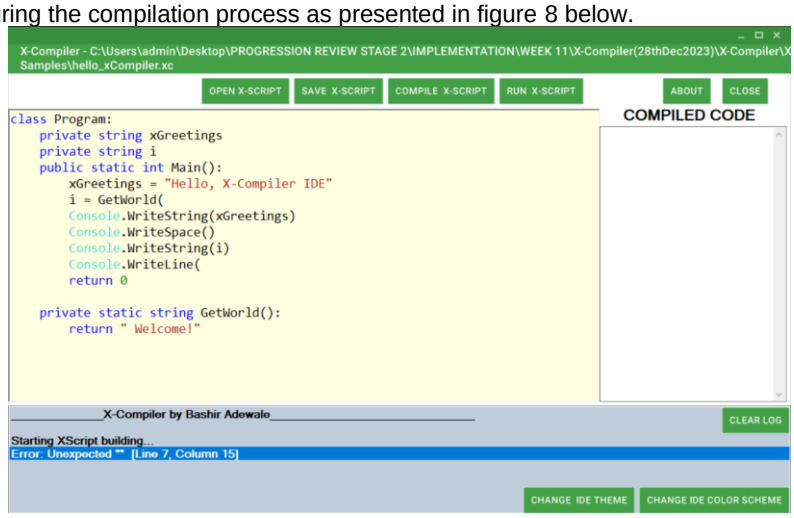


Figure 8: The Erroneous Code GUI.

6. CONCLUSION

Overall, the study on CSXMs contributed to a deeper understanding of concurrent systems, while the design and implementation of the X-compiler showcased the feasibility of creating a synergistic blend of C# and Python programming languages. The results highlight the potential for enhanced language interoperability and the development of efficient compilers that leverage the strengths of multiple programming paradigms. Future work may involve further optimizations using Machine Learning algorithms, language feature enhancements, and real-world applications of the compiled code in diverse computing environments.

REFERENCES

[1] Bohme, M., Cadar, C. and Roychoudhury, A. (2020). Fuzzing: Challenges and Reflections. *Digital Object Identifier*.10.1109/MS.2020.3016773. Page 79-86.

[2] Chaflekar, S. H., Lokhande, A., Gomase, P. and Shinganjude, R. (2017). Compiler Architecture and Design Issues. *International Journal of Computer Science Trends and Technology (IJCST)*. Volume 5. Issue 3.

[3] Chen, Y., Groce, A., Zhang, C., Wong, W. K., Fern, X., Eide, E., and Regehr, J (2013). Taming Compiler Fuzzers. *In PLDI*.pages 197–208.

[4] Chen, J., Hu, W., Hao, D., Xiong, Y., Zhang, H. Zhang, L. and Xie, B. (2016). An Empirical Comparison of Compiler Testing Techniques. *In Proceedings of the 38th International Conference on Software Engineering*. Page 180-190.

[5] Chen, J., Donaldson, A. F., Zeller, A., and Zhang, H. (2017). Testing and Verification of Compilers. *Dagstuhl Reports*. Vol. 7. Issue 12. Page 50-65.

[6] Chen, J. (2018). Learning to Accelerate Compiler Testing. *In Proc. Of the 40th Int'l Conf. on Software Engineering (ICSE 2018)*. Peking University, CN.

[7] Chen, J., Wang, G., Hao, D., Xiong, Y., Zhang, H., and Zhang, L. (2019a). History-guided Configuration Diversification for Compiler Test Program Generation. *In Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. Page 305-316.

[8] Chen, Z., Bihuan, C., Linlin, C., Xin, P., and Wenyun, Z. (2019b). A Large-Scale Empirical Stud of Compiler Errors in Continuous Integration. *In Proceedings of the 27th ACM Joint European Software Engineering Confereence and Symposium on the Foundations of Software Engineering (ESEC/FSE '19*). Tallinn, Estonia. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338917

[9] Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A. Henke, F. W., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H. and Zimmermann, W. (2015). Compiler Correctness and Implementation Verification: The Verifix Approach.*ResearchGate*. https://www.researchgate.net/publication/2274826.

[10] Chen, J., Patra, J., Pradel, M., Xiong, Y., Zhang, H., Hao, D. and Zhang, L. (2020). A Survey of Compiler Testing. *ACM Comput. Surv*. 1, 1, Article 1. Page 1-35. https://doi.org/10. 1145/3363562

[11] Chen, J. and Suo, C. (2022). Boosting Compiler Testing via Compiler Optimization Exploration. *ACM Trans. Softw. Eng. Methodl*. 31, 4 Article 72. Page 1 – 33. https://doi.org/10.1145/3508362

[12] Junjie, C., Wenxiang, H., Dan, H., Yingfei, X., Hongyu, Z., Lu, Z., and Bind, X. (2016). An Empirical Comparison of Compiler Testing Technique. *ICSE '16*, Austin, TX, USA. http://dx.doi.org/10.1145/2884781.2884878

[13] Kossatchev, A. S. and Posypkin, M. A. (2005). Survey of Compiler Testing Methods. *Programming and Computer Software*. Vol. 31, No. 1. Page 10–19.

[14] Le, V., Afshari, M. and Su, Z. (2014). Compiler Validation via Equivalence Modulo Inputs. *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Page 216-226.

[15] Leroy, X. (2009). A Formally Verified Compiler Back-end. *Journal of Automated Reasoning*. **43** (4): Page 363 –446. arXiv:0902.2137. doi:10.1007/s10817-009-91554. ISSN 0168-7433

[16] Marcozzi, M., Tang, Q., Donaldson, A. F. and Cadar, C. (2019). Compiler Fuzzing: How much Does it Matter? *Proc. ACM Program. Lang. 3, OOPSLA*. Article 155. Page 1-29.

[17] Pahade, P. and Dawale, M. (2019). Introduction to Compiler and its Phases. *International Research Journal of Engineering and Technology (IRJET)*. Volume 6. Issue 1. Page 1318-1322.

[18] Phung, K., Jayatilake, D. Ogunshile, E. and Aydin, M. (2021). A Stream X-Machine Tool for Modelling and Generating Test Cases for Chronic Disease Based on State-Counting Approach. *Programming and Computer Software*. Vol. 47. No. 8. Page 765-777. DOI: 10.1134/S0361768821080211

[19] Pizzolotto, D. and Inoue, K. (2021). Identifying Compiler and Optimization Level in Binary Code from Multiple Architectures. *IEEE Access Digital Object Identifier10.1109/ACCESS.2021.3132950*. Volume 9. Page 1- 15.

[20] Sanusi, B. A., Olabiyisi, S. O., Olowoye, A. O. and Olatunji, B. L. (2019). Software Defect Prediction System

using Machine Learning based Algorithms. *Journal of Advances in Computational Intelligence Theory*, 1(3), 1–9. http://doi.org/10.5281/zenodo.35908 41

[21] Sanusi, B. A., Ogunshile, E., Aydin, M., Olabiyisi, S. O., and Oyediran, M. O. (2022). Development of Communicating Stream X-Machine Tool for Modeling and Generating Test Cases for Automated Teller Machine. *Computer Science & Information Technology (CS & IT) – CSCP*. DOI: 10.5121/csit.2022.121407. Page 77 – 90.

[22] Schumi R, Sun J. SpecTest (2021). Specification-Based Compiler Testing. *Fundamental Approaches to Software Engineering*. 2021 Feb 24;12649:269–91. doi: 10.1007/978-3-030-71500-7_14. PMCID: PMC7978860.

[23] Simons, A.J.H. and Leftticaru, R. (2020). A Verified and Optimized Stream X-Machine Testing Method, with Application to Cloud Service Certification. *Software Testing, Verification and Reliability.* 30(3): e1729. https://doi.org/10.1002/stvr.1729

[24] Sun, C., Le, V., Su, Z.(2016). Finding and analyzing compiler warning defects. *In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016* Austin, TX, USA, May 14-22, 2016. pp. 203 213. https://doi.org/10.1145/2884781.2884879

[25] Tang, Y., Jiang, H., Zhou, Z., Li, X., Ren, Z. and Kong, W. (2020). Detecting Compiler Warning Defects Via Diversity-Guided Program Mutation. IEEE Transactions on Software Engineering. **DOI:** 10.1109/TSE.2021.3119186

[26] Uwe, S. (2011). The Real Story Behind the Java 7 GA Bugs Affecting Apache Lucene / Solr. https://blog.thetaphi.de/2011/07/real-story-behind-java-7-ga-bugs.html. Bremen, Germany.

[27] Yang, X., Chen, Y., Eide, E. and Regehr, J. (2011). Finding and Understanding Bugs in C Compilers. *In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. Page 283–294.