VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINERING

GRADUATION THESIS

PROGRAM ANALYSIS USING STATIC METHODS

COMMITTEE	: **COMPUTER SCIENCE**
SUPERVISOR	: **Prof. THO, QUAN THANH**
EXAMINER	: …

---o0o---

STUDENT	: **MAI, DINH HOANG (50901524)**

HO CHI MINH CITY, DECEMBER 2013

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Prof. Tho T. Quan, who gave me a clear direction for my research work and led me to complete this graduation thesis. His huge knowledge, kindness and sense of humor impress and educate me all the time. With a great relationship and network with foreign countries, he brought us exchange programs with other countries as well as chances to get scholarship to study higher.

I would also like to thank my mentor, Mr. Hai M. Nguyen, a Ph. D. student at Ho Chi Minh City University of Technology (HCMUT), for directly guiding me during my research period. An opportunity to participate in a Ph. D. project (CFG Reconstruction From Binary Code Using A Hybrid Approach) gave me a great chance to extend my knowledge. Throughout the time working with him, currently I have a strong background research about symbolic execution techniques and its applications.

Moreover, I would like to thank my other mentor, Mr. Long H. Pham, a M.S. student at HCMUT. Mr. Long helped me research and solve many problems about fault localization techniques. The presentation at the 2nd International Engineering and Technology Education Conference (IETEC 2013) and a publication paper in there are greatly successful results for an undergraduate student when I have a chance to work with him in the Group Testing project.

Furthermore, I am grateful to Mr. Duy T. B. Le from Singapore Management University (SMU), who helped me a lot when I started to research the field of fault localization. I would like to express my appreciation to Dr. Phung H. Nguyen, Dr. Thang H. Bui, Mr. Thuan D. Le, Mr. Binh T. Ngo, Mr. Anh D. Le and Ms. An N. Nguyen (HCMUT), who shared their time and knowledge to help me in my research and relevant projects.

I appreciate my good friends, Mr. Nam P. Mai and Mr. Giang V. Trinh, for working together with me in same projects, and Mr. Tai T. Huynh, Mr. Huy D. Nguyen, Mr. Vinh Q. Vu, Mr. Tinh V. Nguyen, Mr. Huy B. Nguyen, Mr. Nam H. Lai, Mr. Quy D. Vu, Ms. Tam T. Nguyen, Ms. Ngoc B. Tran for sharing good time throughout these days.

Finally, I would like to express a great love and appreciation to my family for supporting and encouragin me during my works.

ABSTRACT

TÓM TẮT LUẬN VĂN

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

1.1. OVERVIEW

In the current technology age, software is widely used in almost all fields of life. The number of newly written programs is increasing dramatically in both quantity and complexity aspects under latest advanced techniques, but they are still far from bug-free. In the software development process, software quality insurance plays a greatly important role. The problems are how to ensure the correctness of the program, how to prevent the program compromising the system security, how to show the reasons why the program is wrong, how to locate exactly the lines of code in the program containing bugs and even more how to automatically debugging.

A primary method used in industry is *Software Testing* [1] that based on program analysis techniques to construct the set of test data (test cases), and then these test cases are used to execute the program for testing. Despite currently having a widespread usage, *Software Testing* still face to several obstacles.

- The execution of program on test suite cannot totally prove the correctness of program. The main cause is due to the test case generation process not covering all cases in program.
- It also takes much time and efforts to build up an environment for running the software in practice due to the technology requirement of that software, especially with the software using new technology.

To overcome such problems, recent works on software verification using static methods show great promise and appear to be the ultimate techniques to confirm the program‘s correctness and prevent from compromising the system security. One of the main feature of static methods is that it can check a program but no need to execute that program actually. In addition, the checking process is independent of the technology as well as the programming language using in the software. Thus, static methods can overcome above issues.

Recently, two static methods of *Theorem Proving* and *Model Checking* are suggested for program testing. They use mathematics-based techniques to check program’s properties without actually running programs. While *Theorem Proving* can verify the program’s correctness, *Model Checking* can generate counter examples to help trace down the bugs via corresponding execution flows if the program is false. Currently, these methods are combined to build a web-based tutoring system called *Prove System*, which is an automated assessment system for improving programming skills of students. This report presents the specification of this *Prove System* that strictly relating to two static methods of *Theorem Proving* and *Model Checking* as well as the improvement of these methods for upgrading the *Prove System*.

Symbolic Execution [2] is another technique that program is symbolically executed instead of running on a set of sample test cases, which can be an extreme method for program proving, program testing as well as program debugging. From a simple view, *Symbolic Execution* is an enhanced testing technique because when a program is symbolically run, it is equivalent to a large number of normal test cases. This result can be checked against the programmer's expectation for correctness either formally and informally. This report describes a practical approach of test case generation using *Symbolic Execution*.

The class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs.

- If the control flow of the program is completely independent of the input variables, a single symbolic execution will suffice to check all possible executions of the program.
- If the control flow of the program is dependent on the inputs, one must resort to a case analysis. Often the set of input classes needed to exhaust all possible cases is practically infinite, so this is still basically a testing methodology. However, the input classes are determined only by those inputs involved in the control flow, and symbolic testing promises to provide better results more easily than normal testing for most programs.

Moreover, *Fault Localization* techniques show impressive results of specifically finding bugs in program that help to cut down a great expenditure and the time consuming in the debugging process. An essential requirement for *Fault Localization* techniques to locate accurately the errors in program is the input test suite. A test suite of full coverage in program can be generated using *Symbolic Execution* techniques and *SMT Solvers*. Thus, *Fault Localization* and *Symbolic Execution* can combined to become powerful techniques for program testing. This combination is also presented in this report through a new framework of *Prove System* mentioned above.

1.2. THESIS PURPOSE

The aim of this thesis is to present techniques in the program analysis process using static methods:

- Using *Symbolic Execution* technique with related algorithms to generate an excellent test data for program testing. For each kinds of programming languages, *Symbolic Execution* will be simulated on an intermediate representation to create path conditions. After that, a *SMT Solver* will calculate and solve these path conditions to give test cases.
- Using *Symbolic Execution, Fault Localization* and other related boosting techniques to build a new framework of an automated assessment system. This system will be present as a web-based tutoring tool for improving programming skills for students.

1.3. CONTRIBUTION

The main contributions are strictly related to two projects:

- In the Ph.D. project called *CFG Reconstruction From Binary Code Using A Hybrid Approach* [3]
    - We apply *Symbolic Execution* into the *Static Analysis* phase to generate appropriate test data when indirect jumps and calls occur.
    - We also suggest solutions of the invariant generation problems for the linear system.
- In the existing project of our group called *Prove System*, we upgrade it to a new version (version 9.0) with several improvements as follow:
    - Implement a new framework using *Symbolic Execution* and *Fault Localization* for simulating and checking student's program.
    - Improve the current test case generation technique by implementing a highly efficient constraint-based algorithm called *CTG^E* [4] and some boosting techniques such as *Bound Analysis* and *Interpolation* for tackling the path-explosion problem [5].
    - Implement a module that can check two versions of a program.
    - Enhance the current website with the latest web technology and a more interactive interface through a feedback system.
    - Build up a server for *Prove System* running as a virtual machine that can be flexible to replace or improve any system resources.

1.4. PROJECT SCOPE

In the *CFG* project, we will look at X86 instructions from each category below:

- Data movement instructions: mov, push, pop, lea.
- Arithmetic and logic instruction: add, sub, inc, and, or, xor, not, neg, shl, shr.
- Control flow instructions: jmp, je, jne, jz, jg, jge, jl, jle, cmp, call, ret.

In the *Prove System* version 9.0, we focus on programs writing in C programming language. We still have not tackled some complex instructions such as pointer, calling procedure, etc. The programs we use for experiments are only at student level, but there are still many techniques we can apply, so its future is very encouraging.

1.4. THESIS STRUCTURE

The remainder of this report follows this structure. The next Section 2 contains the preliminary knowledge that required to understand the system including symbolic execution, test case generation, X86 binary structure and fault localization.

After that, in Section 3, we concern a test case generation technique for binary code using *Symbolic Execution*. At this stage, we will introduce the main features and the problems of the CFG project, as well as the solution based on *Symbolic Execution* technique.

Furthermore, in the Section 4 we will present a new framework (version 9.0) of the existing *Prove System* (version 8.0) for students to improve programming skills. We will talk about the process of system implementation and working flow in this new version.

We do experiments on the system in Section 5 with screenshots and reviews. Section 6 describes our conclusions and future works. Section 7 contains the reference materials. The appendixes are showed in Section 8.

2. PRELIMINARY KNOWLEDGE

2.1. SYMBOLIC EXECUTION

Definition

Example

Symbolic execution is a practical approach for proving correctness of programs. Unlike normal execution, programs will execute with symbols as arguments.

Forward, Backward, Bound analysis

…

2.2. TEST CASE GENERATION

Definition

Example

CTG-E Algorithm is an effective constraint-based test case generation algorithm for detecting regression bugs in evolving programs

Boosting concolic testing via Interpolation

…

2.3. FAULT LOCALIZATION

Definition

Example

Spectrum-based Fault Localization is one kind of automated debugging techniques that effectively narrows down the possible locations of software faults and thus helps save developers' time. This type of techniques usually takes in a set of program execution profiles along with their labels (i.e., passed or failed executions), and recommends suspicious program elements to developers for manual inspection.

Spectrum-based fault localization: group testing, tarantula, ochiai

Visualization test information to assist fault localization

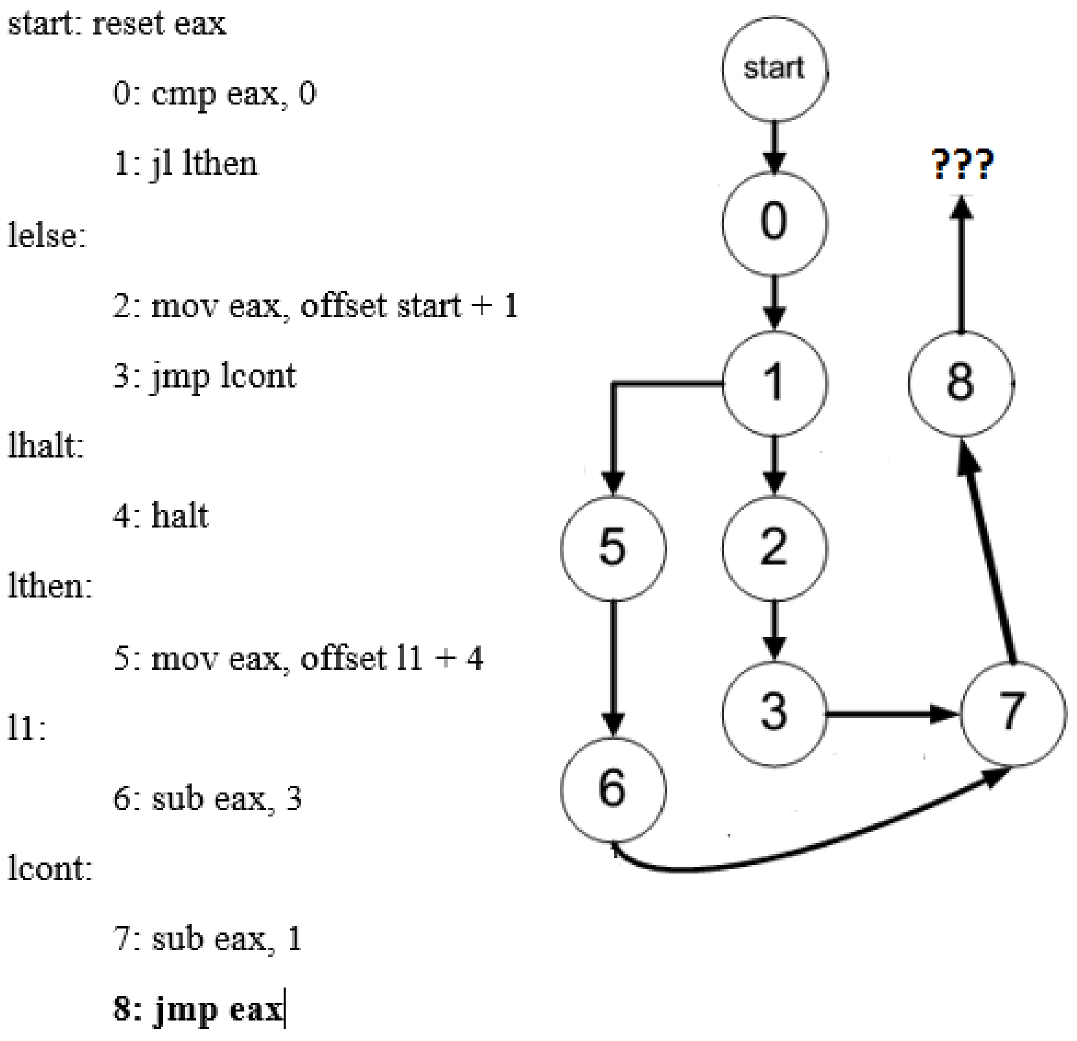
…

3. A TECHNIQUE OF TEST CASE GENERATION FOR BINARY CODE USING SYMBOLIC EXECUTION

3.1. PROBLEMS ANALYSIS AND SOLUTIONS

In the Ph.D. project *called CFG Reconstruction From Binary Code Using A Hybrid Approach*, it discusses a hybrid approach of combining *Static Analysis* and *Dynamic Testing* to construct CFG from binary code. The problem occurs when processing the indirect jumps during the CFG construction process that can produce false targets. The detailed specification of this project is demonstrated in Appendix A.

In this report, we apply *Symbolic Execution* techniques into the static analysis phase to create the path conditions for each path executions and to generate test cases when reaching an indirect jump instruction.



```
start: reset eax
        0: cmp eax, 0
        1: jl lthen
lelse:
        2: mov eax, offset start + 1
        3: jmp lcont
lhalt:
        4: halt
lthen:
        5: mov eax, offset l1 + 4
l1:
        6: sub eax, 3
lcont:
        7: sub eax, 1
        8: jmp eax
```

Figure 1: A binary code example with an indirect jump at line 8

Figure 1 shows an example of binary code that contains an indirect jump and a CFG constructed from this code. The instructions at line 1 and line 3 are direct jump instructions because they jump directly to fixed addresses (label *lthen* and *lcont*), but the instruction at line 8 is an indirect jump instruction because it jumps to the address which is the value of the register *eax*. The value of the register *eax* can change at any time during the program execution, so the instruction at line 8 can jump to any addresses. Thus, the CFG construction stops at the step 8 when encountering an indirect jump instruction.

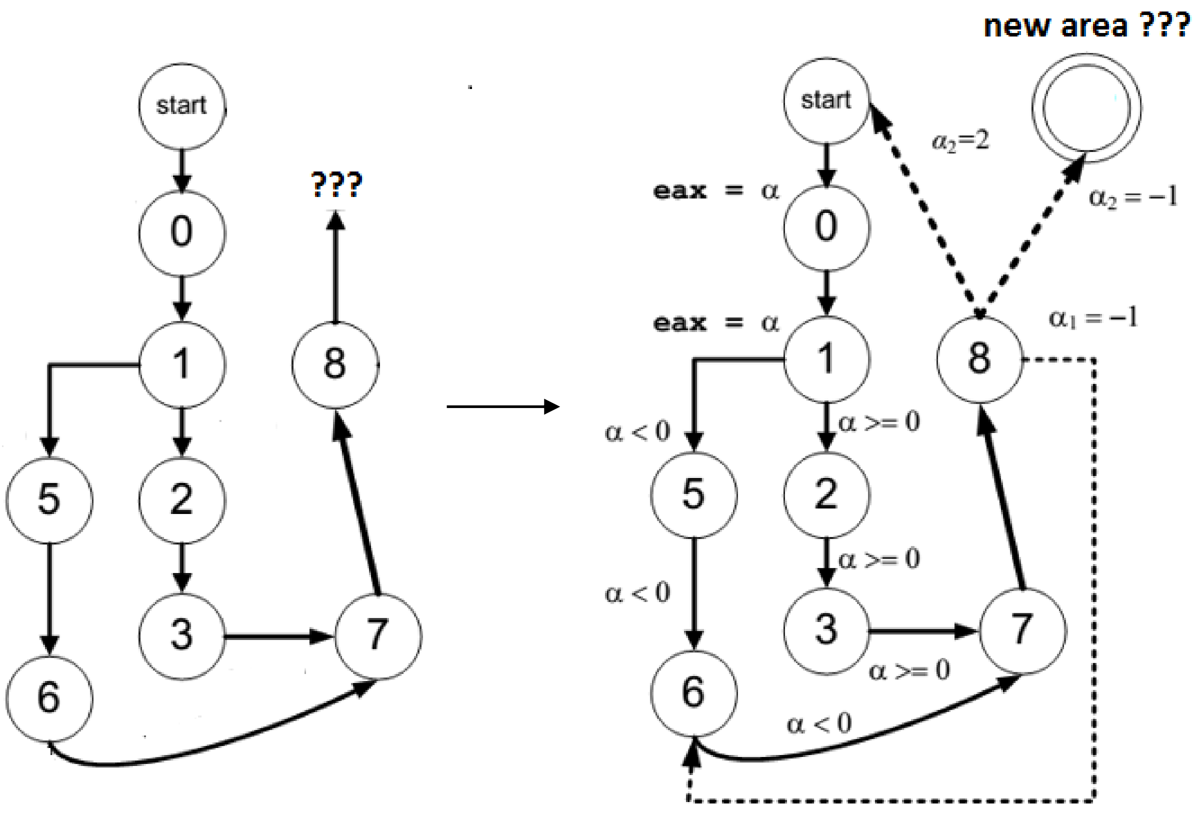


Figure 2: The CFG constructed after the static analysis phase

To continue the CFG construction process, we enter into the Static Analysis phase. In this phase, we use a static method of *Symbolic Execution* [2] by running this program with symbolic values to create path conditions corresponding to each execution path of the program. It can be clearly seen from Figure 2, the instruction at step 1 is a branch instruction with two execution paths, so we apply *Symbolic Execution* techniques to get the required conditions to continue running through each paths.

At the step 8 of Figure 2, this is an indirect jump instruction, we already have all execution paths for reaching this instruction. In this example, we have two paths: $P_1 = (start \rightarrow 0 \rightarrow 1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8)$ and $P_2 = (start \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7 \rightarrow 8)$. Given that the initial value of register *eax* is α, the path conditions of P_1 and P_2 can be evaluated as ($\alpha < 0$) and ($\alpha >= 0$) respectively. With the path condition of each execution path to this indirect jump instruction, we use the *Z3 SMT Solver* to solve these constraints to generate test cases for the next *Dynamic Analysis* phase.

At this time, with these test cases, the *Dynamic Analysis* phase will be executed. At the indirect jump instruction, it can be jump to an area which we already know before, or it can be jump to a new area which has not been explored yet. If it can reveal a new area, the *Static Analysis* phase will be invoked again with this newly-discovered region and this process will repeat until no new area is discovered.

In term of the static method of *Symbolic Execution*, it is a natural extension of normal execution, providing the normal computations as a special case. Computational definitions for the basic operators of the language are extended to accept symbolic inputs and produce symbolic formulas as output. Thus, each symbolic execution result may be equivalent to a large number of normal test cases and we can generate test cases that can cover all the possible execution paths of the program.

However, a major challenge of binary code analysis is the loop invariant generation problem. When running the program by *Symbolic Execution* technique, it is absolutely crucial to find conditions to exit the loop for continuing the execution. Currently, invariant generation is still a problem without any ideal solutions. To overcome this problem, several methods have been used such as *Karr* [6], *abstract interpretation by Cousot and Cousot* [7], *Farkas' Lemma* [8], *Mathematical Programming* [9] for generating linear invariants and *Grobner Bases* [10], *Craig Interpolation* [11] for generating non-linear invariants. In this report, we suggest the *Farkas' Lemma* as a sound and complete method for solving linear invariant. We give some concrete examples and the plan to implement this method into the project.

3.2. SYSTEM DESIGN AND IMPLEMENTATION

3.3. REVIEW AND EVALUATION

4. A NEW FRAMEWORK FOR IMPROVING PROGRAMMING SKILLS USING STATIC METHODS

4.1. PROBLEMS ANALYSIS AND SOLUTIONS

The existing project of our group called *Prove System* is an automatic verification system for programming exercises. The purpose of *Prove System* is to provide a web-based tutoring system [12] that can play the role of teacher for teaching programming to students. In this system, static methods are proposed to verify programs' correctness statically without actually running programs. The version 8.0 of *Prove System* released in 2009 has not been updated for about three years. For this reason, we upgrade the *Prove System* to version 9.0 with significant improvements using new algorithms as well as building a more flexible system running below.

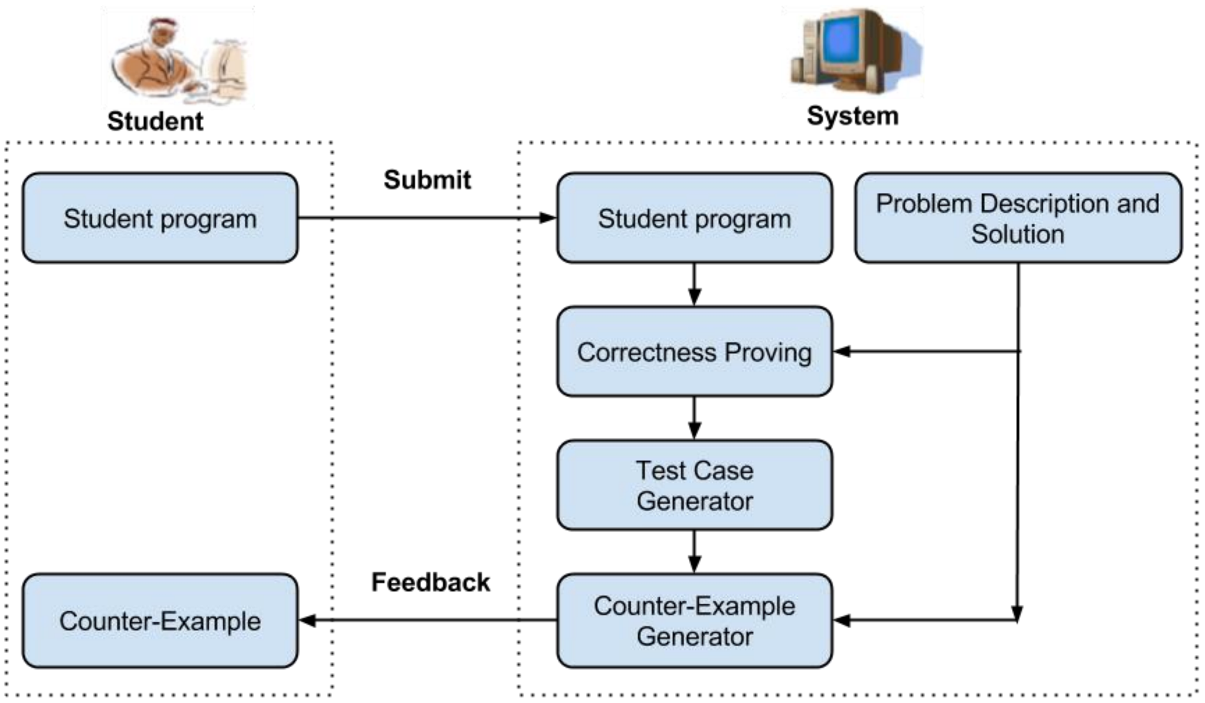
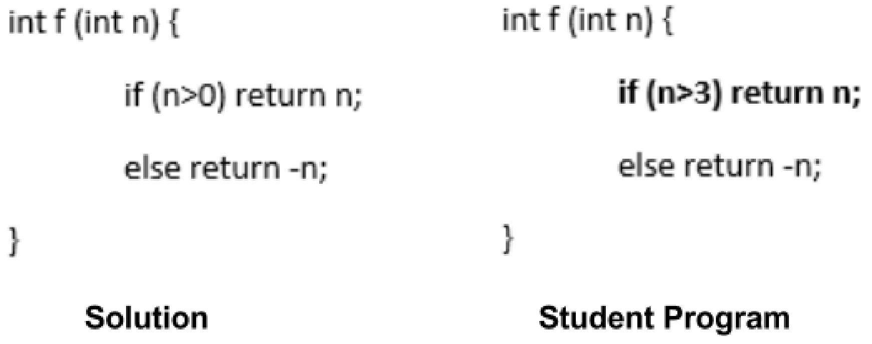


Figure 3: A short description of Prove System version 8.0

Figure 3 shows a short description of the *Prove System* version 8.0. The detailed specification of this system is describes in Appendix B. In this system, some programming exercises are predefined and student can choose to implement any exercises through a web-based interactive system. After submitting the program, the student's program is sent to the *Correctness Proving* module. With the problem description, *Theorem Proving* can verify the correctness of the student program by using *Provers* such as Z3 [13], Simplify [14], Coq [15], Isabelle [16], Alt-Ergo [17] and Redlog [18]. If the result is valid, the system will immediately feedback to student that the program is right. If the result is invalid or unknown, the program will be sent to the next phases.

In the *Test Case Generator* module, the system divides the input space of student program into sub-domains, and then takes samples such that the samples cover all of sub-domains. The sub-domains are decided based on program inputs and flows. In the result, random-generated test cases that can cover all possible paths in student program are produced. Next, in the *Counter-Example Generator* module, a *Model Checker* called *Spin* [19] is used to simulate the program with previous test cases. If the system can conduct a counter-example, the student will receive a counter-example with the tracing path. With this information, student can review the code and find out the error in the program.



```
int f (int n) {
        if (n>0) return n;
        else return -n;
}
```

```
int f (int n) {
        if (n>3) return n;
        else return -n;
}
```

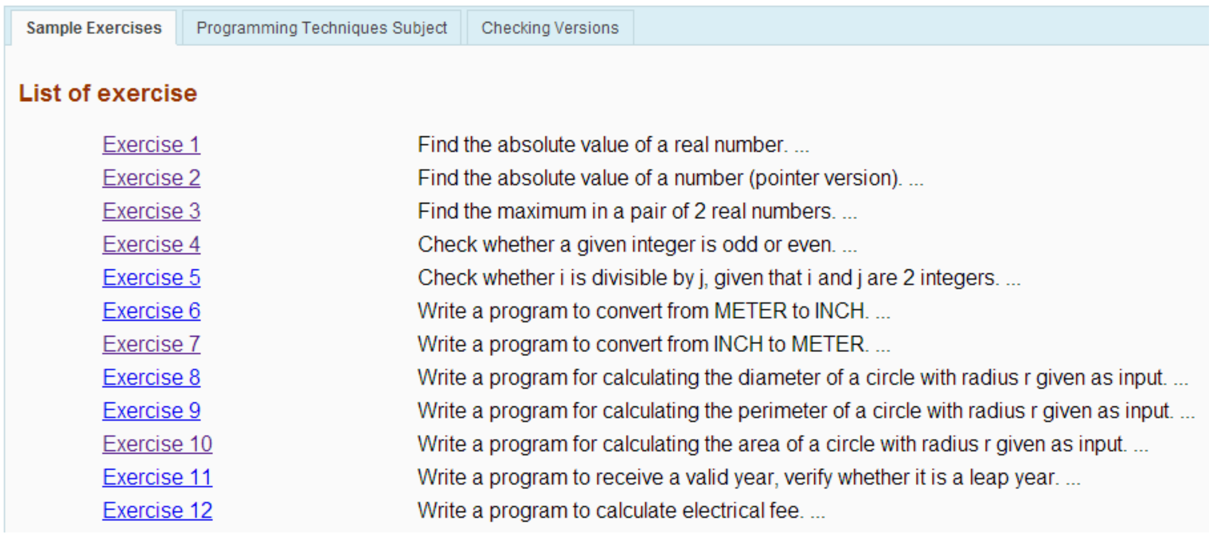
Figure 4: A programming exercise of finding the absolute value of a number

A major limitation of the Prove System version 8.0 is the random test case generation technique. Figure 4 presents an example of this problem. It can be easily seen that the student program is a wrong program and any value in the domain *(0,3)* is caused to a wrong result. According to the algorithm of the Prove System, there are two sub-domains that are *(-∞,3)* and *(3,+∞)* in the program of student, and then test cases are random-generated in these domains. To find a counter-example, the system need to find a test case having a value in domain *(0,3)*, but it is inevitable that in some cases, it can not find a test case like that because of the test case random-generated technique.

Thus, we implement the *Efficient Constraint-based Test Case Generation* algorithm called *CTG*^{E} that can combine all domains of both student program and solution program [4] to produce path conditions that cover all possible execution paths of both programs. Next, we use the *Z3 SMT Solver* to solve these path conditions to generate test cases. To be more effective, we also generate bounded test cases of conditions of the program. At this time, we have a test suite containing full-coverage test cases that cover all possible cases of both student and solution programs and the problem in Figure 4 can be easily solved.

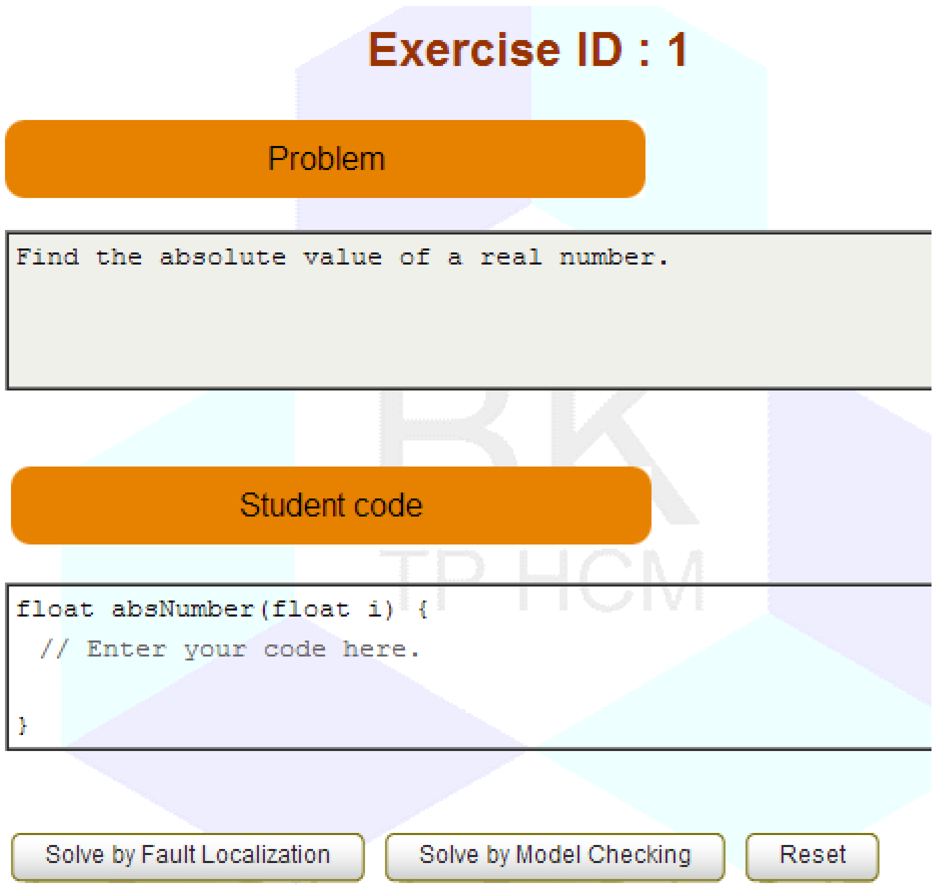
However, one of its major limitations is that there are in general an exponential number of paths in the program to explore, resulting in the so-called path-explosion problem. Recently, several methods have been proposed to tackle this problem such as using *interpolation for pruning paths* [5], *heuristics focused on branch coverage [20], function summaries* [21], *using static/dynamic program analysis* [22] and etc. In this report, we suggest the method based on *interpolation* that significantly mitigates path-explosion by pruning a potentially exponential number of paths that can be guaranteed to not encounter a bug.

4.2. SYSTEM DESIGN AND IMPLEMENTATION



List of exercise

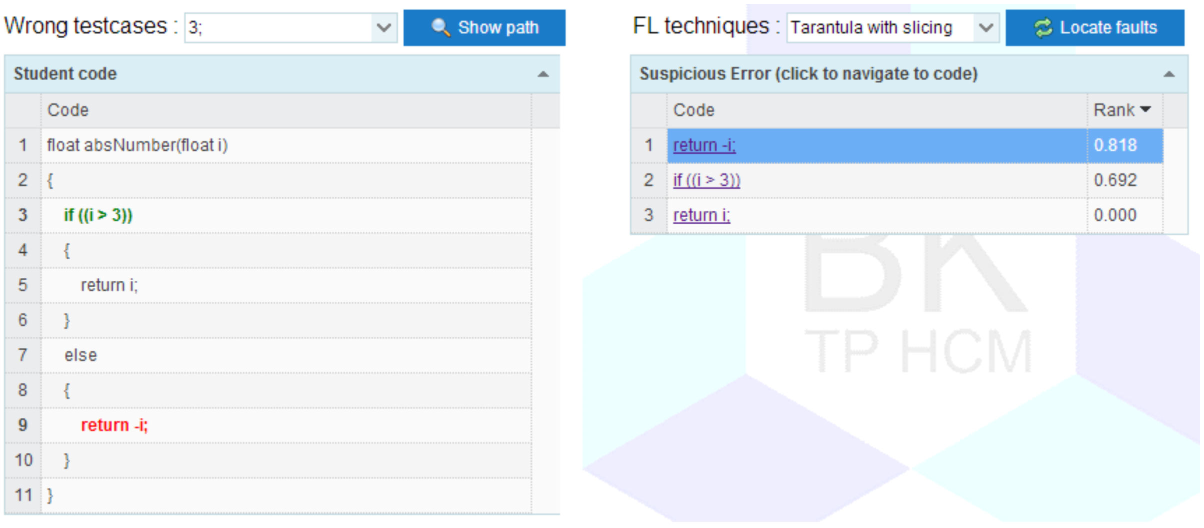
Exercise 1	Find the absolute value of a real number. ...
Exercise 2	Find the absolute value of a number (pointer version). ...
Exercise 3	Find the maximum in a pair of 2 real numbers. ...
Exercise 4	Check whether a given integer is odd or even. ...
Exercise 5	Check whether i is divisible by j, given that i and j are 2 integers. ...
Exercise 6	Write a program to convert from METER to INCH. ...
Exercise 7	Write a program to convert from INCH to METER. ...
Exercise 8	Write a program for calculating the diameter of a circle with radius r given as input. ...
Exercise 9	Write a program for calculating the perimeter of a circle with radius r given as input. ...
Exercise 10	Write a program for calculating the area of a circle with radius r given as input. ...
Exercise 11	Write a program to receive a valid year, verify whether it is a leap year. ...
Exercise 12	Write a program to calculate electrical fee. ...



Find the absolute value of a real number.

```
float absNumber(float i) {
  // Enter your code here.

}
```

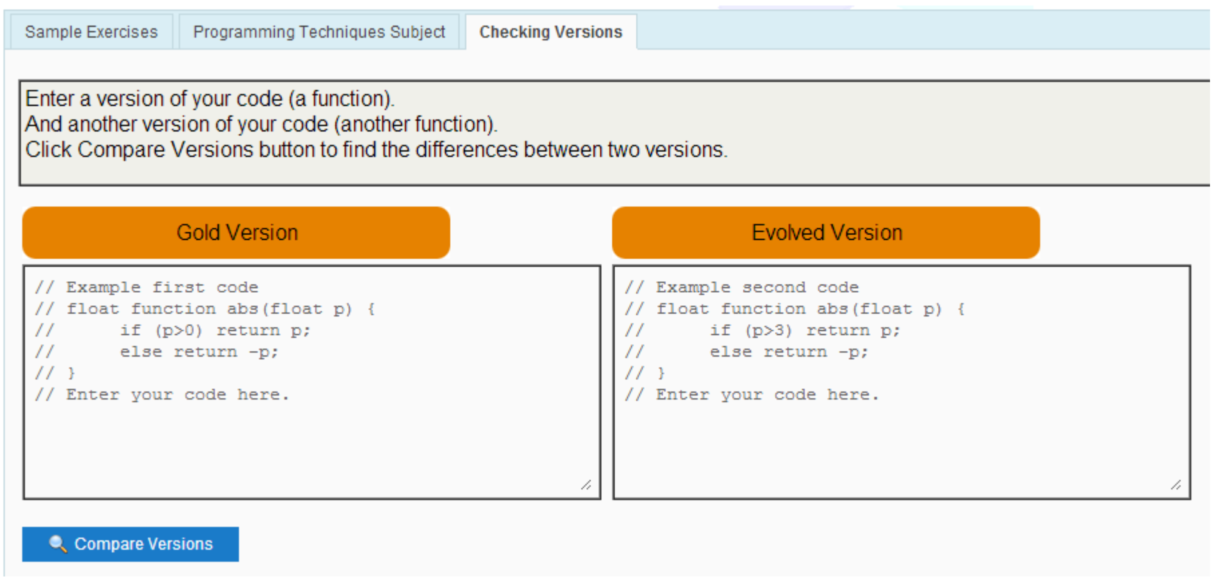


Student code

```
float absNumber(float i)
{
    if ((i > 3))
    {
        return i;
    }
    else
    {
        return -i;
    }
}
```

Suspicious Error (click to navigate to code)

	Code	Rank
1	return -i;	0.818
2	if ((i > 3))	0.692
3	return i;	0.000



Enter a version of your code (a function).
And another version of your code (another function).
Click Compare Versions button to find the differences between two versions.

```
// Example first code
// float function abs(float p) {
//     if (p>0) return p;
//     else return -p;
// }
// Enter your code here.
```

```
// Example second code
// float function abs(float p) {
//     if (p>3) return p;
//     else return -p;
// }
// Enter your code here.
```

Figure 5: Checking versions module

Last but not least, we also implement a new module that can compare two versions of a student's program and show test cases having different results in two program versions that can be seen from the Figure 5. The main idea is taken from the test case generation technique based on *CTG*^{E} algorithm mentioned above. We also combine all domains of both gold version and evolved version, then running through a *Simulator*.

4.3. REVIEW AND EVALUATION

5. EXPERIMENTS

5.1. CFG RECONSTRUCTION FROM BINARY CODE USING A HYBRID APPROACH

5.1.1. INPUT/ OUTPUT

…

5.1.2. TEST CASES

…

5.1.3. SCREENSHOT

…

5.1.4. REVIEW

…

5.2. PROVE SYSTEM VERSION 9.0

5.2.1. INPUT/ OUTPUT

…

5.2.2. TEST CASES

…

5.2.3. SCREENSHOT

…

5.2.4. REVIEW

…

6. CONCLUSIONS AND FUTURE WORKS

Predict bug-fixing time

Path explosion

a framework for automatic verification of programming exercises called prove system is implemented and employed in our real education university environment with several impressive results although it still has some limitations

7. REFERENCES

[1] A. Spillner, T. Linz, and H. Schaefer, “Software Testing Foundations. dpunkt. verlag,” *Heidelberg, Ger.*, 2006.

[2] J. C. King, “Symbolic Execution and Program Testing,” 1976.

[3] M. H. Nguyen, H. City, T. B. Nguyen, T. Quan, and M. Ogawa, “A Hybrid Aproach for Control Flow Graph Construction from Binary Code.”

[4] A. D. Le, T. T. Quan, N. T. Huynh, and P. H. Nguyen, “: an effective constraint-based test-case generation algorithm for detecting regression bugs in evolving programs,” 2008.

[5] J. Jaffar, V. Murali, and J. a. Navas, “Boosting concolic testing via interpolation,” *Proc. 2013 9th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2013*, p. 48, 2013.

[6] M. Karr, “Affine relationships among variables of a program,” *Acta Informatica*, vol. 6. pp. 133–151, 1976.

[7] “Abstract interpretation - a unified lattice model for static analysis of programs by construction or approximation of fixpoints.pdf.” .

[8] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998, p. 484.

[9] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Scalable Analysis of Linear Systems using Mathematical Programming.”

[10] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, “Non-linear Loop Invariant Generation using Gr obner Bases,” 2004.

[11] J. Esparza, S. Kiefer, and S. Schwoon, “Abstraction Refinement with Craig Interpolation and Symbolic Pushdown Systems.”

[12] SAVE, “Prove System,” 2013. [Online]. Available: http://elearning.cse.hcmut.edu.vn/provegroup/.

[13] “Z3 - An Efficient SMT Solver.” [Online]. Available: http://z3.codeplex.com/.

[14] G. Nelson, J. B. Saxe, and D. Detlefs, “Simplify: a theorem prover for program checking,” *Journal of the ACM*, vol. 52. pp. 365–473, 2005.

[15] G. Huet, G. Kahn, and C. Paulin-Mohring, “The Coq Proof Assistant A Tutorial,” *Rapp. Tech.*, vol. 178, pp. 1–44, 2007.

[16] “Isabelle.” [Online]. Available: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[17] “Alt-Ergo Theorem Prover.” [Online]. Available: http://alt-ergo.lri.fr/.

[18] “Redlog.” [Online]. Available: http://redlog.dolzmann.de/.

[19] “Spin - Formal Verification.” [Online]. Available: http://spinroot.com/spin/whatispin.html.

[20] J. Burnim and K. Sen, “Heuristics for Scalable Dynamic Test Generation,” *2008 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008.

[21] P. Godefroid, “Compositional dynamic test generation,” *ACM SIGPLAN Notices*, vol. 42. p. 47, 2007.

[22] P. Boonstoppel, C. Cadar, and D. Engler, “RWset: Attacking path explosion in constraint-based test generation,” in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS’08)*, 2008, vol. 4963, pp. 351–366.

[23] L. H. Pham, N. P. Mai, M. H. Dinh, T. T. Quan, and H. Q. Ngo, “Assisting Students in Finding Their Own Bugs in Programming Exercises using Verification and Group Testing Techniques,” 2013.

[24] J. A. Jones and M. J. Harrold, “Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique Categories and Subject Descriptors.”

[25] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of Test Information to Assist Fault Localization.”

[26] L. Gong, D. Lo, L. Jiang, and H. Zhang, “Interactive fault localization leveraging simple user feedback,” *2012 28th IEEE Int. Conf. Softw. Maint.*, pp. 67–76, Sep. 2012.

8. APPENDIXES

8.1. APPENDIX A: THE PROJECT OF CFG RECONSTRUCTION FROM BINARY CODE USING A HYBRID APPROACH

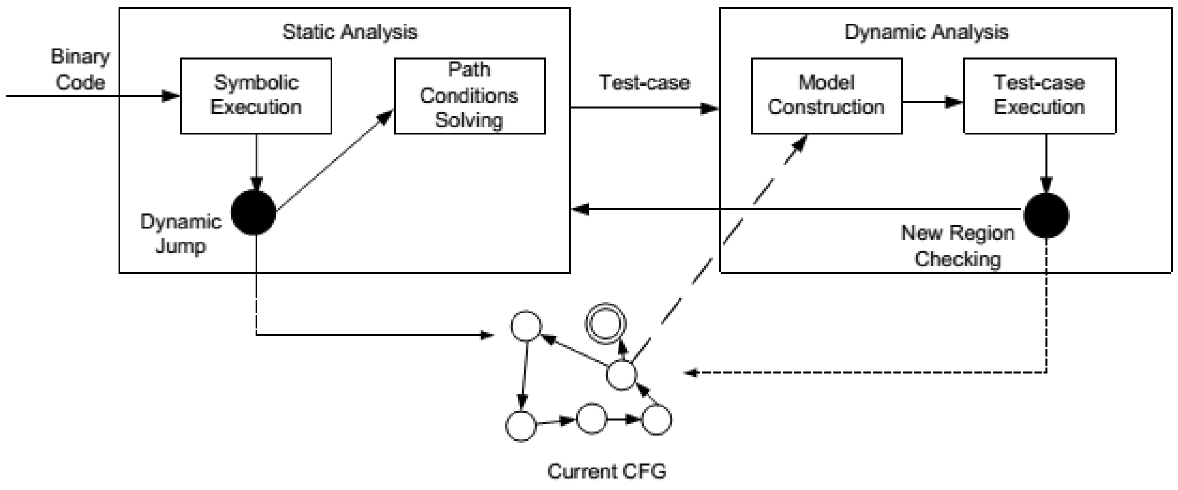


Figure 6: The framework of combining static analysis and dynamic analysis for CFG reconstructed from binary code

8.2. APPENDIX B: THE PROVE SYSTEM VERSION 8.0

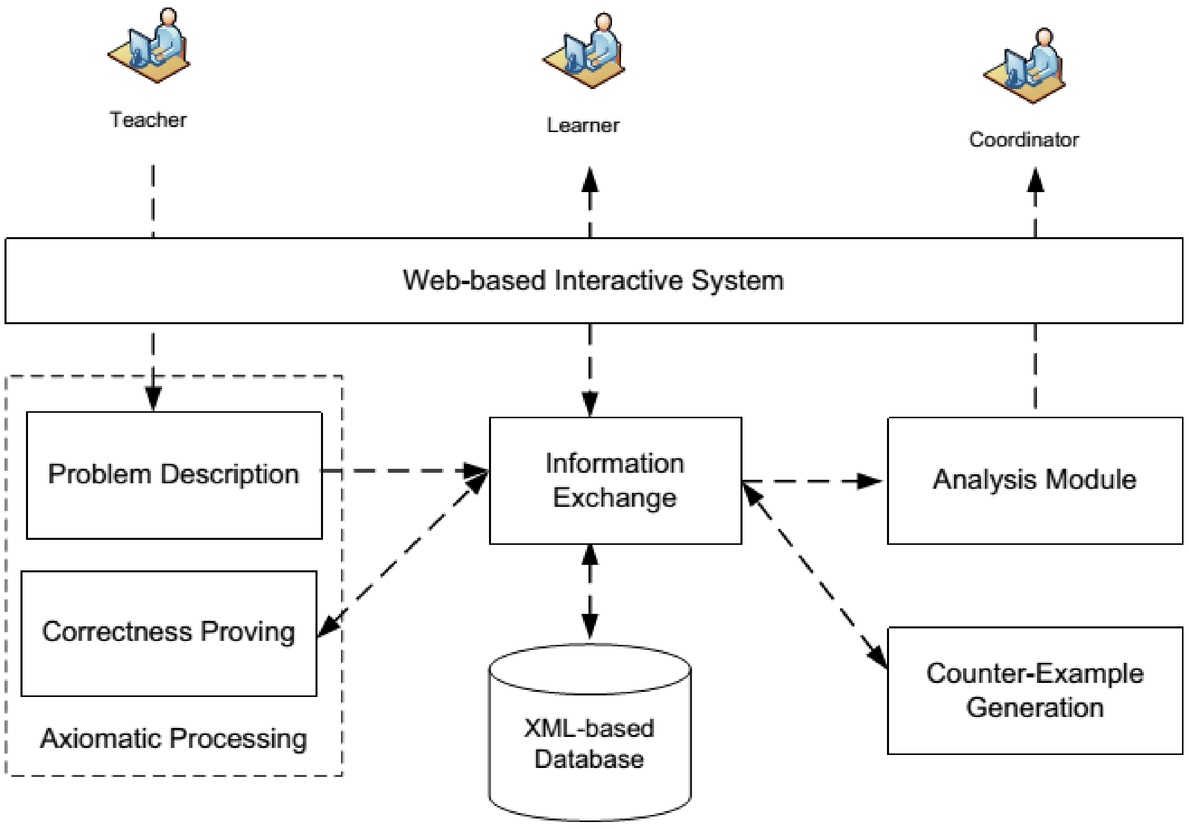


Figure 7: A framework for automatic verification of programming exercises