iiOSProTrain: An Interactive Intelligent Online System for Programming Training

Tho T. Quan, Phung H. Nguyen, Thang H. Bui, Thuan D. Le, An N. Nguyen, Duc L.N Hoang, Vu H. Nguyen, Binh T. Nguyen
Hochiminh City University of Technology, Vietnam
Email: {qttho, phung, thang, thuanle}@cse.hcmut.edu.vn, {annguyen2210, hoanglenghiaduc }@gmail.com, huuvu198x@yahoo.com, binhnt@3forcom.com

***Abstract*— Programming is a crucial skill which is required to be mastered for students on all disciplines of the Computer Science field. For novice students, they always desire explanation and help for all errors encountered in their own solutions. With the recent advancement of Internet technologies, online tutoring systems are increasingly considered. Various systems and applications have been introduced for teaching programming. In this paper, we introduce a tutoring system known as iiOSProTrain (interactive intelligent Online System for Programming Training) for teaching students programming. In iiOSProTrain, we employ two popular formal methods, which are *theorem proving* and *model checking*, for verifying students' works and giving them detailed feedback. The usage of formal methods renders our system the following advantages, as compared to similar systems. First, iiOSProTrain can give an absolute confirmation on the correctness of the submitted programs. More importantly, iiOSProTrain adopts the concept of *structured error-flow* to give students traceable feedbacks on their mistakes, thus allowing them to track and correct the logic errors in a flexible and convenient manner. Currently, iiOSProTrain is deployed at Faculty of Computer Science and Engineering, Ho Chi Minh University of Technology, Vietnam for teaching first year students on programming methodologies.**

***Index Terms*— intelligent tutoring systems, programming training, formal methods, program verification, courter-examples, error-flows**

I. Introduction

As stated in ACM Computing Curricula 2001 [1], programming-involved courses are regarded as the basis of most of the Computer Science (CS) studies. In the other words, possession of good programming skill is necessary to secure the learning outcomes in this field. In the past, programming is mostly taught in the traditional face-to-face basis. Recently, Web-based tutoring systems that can play the role of teacher for training programming, partially or completely, are increasingly considered. With the convenient communication offered by the Internet, such systems can render an interactive environment. The concept of e-learning on a Web-based plafform combined with computer science technologies also offer a promising environment for other domains, such as Anti-Money Laundering Training in international banks [2].

Traditionally, the interaction required for an education environment should fulfill the following requirements [3]: (i) presence of individuals; (ii) presence of some action or incident between them; and (iii) impact on the intrinsic or extrinsic condition of the individuals. Whereas the two first requirements are covered by almost Web-based tutoring systems, the third requirement still poses a challenge to be tackled effectively.

In the context of this paper, the impact imposed on the individuals involving in a tutoring system that we want to focus on is the improvement on programming skill that the learners acquire. For the mastery of programming skill, solid background in theory and profound experience in practical exercises are both required [4], as it well serves the objective of helping students to gain explicit information beyond the experiential implicit knowledge [5]. Nevertheless, it is undeniable that one of the best methods to acquire and sharpen programming skill is to practice with exercises of substantial size [6].

Research has shown that misconceptions among students can be eliminated when formative feedback and correct solutions can be obtained [7]. Especially, analyzed explanations on the submitted programs can help students become more effective programmers [8]. Also, it would help students get a better understanding of the exercise requirements and have a better chance of supplying the correct answer [9]. However, providing timely and useful feedback on student programming exercises is a difficult and time-consuming task, especially with the large class sizes [10].

In traditional education of programming training, this problem is countered by manual review and inspection offered by class tutors [11]. However, it is not efficiently practical when dealing with large numbers of students and practice exercises. To automate the inspection process, one popular method suggested is generating a suit of test cases based on the coverage analysis [12] and executing the programs with all of the test cases. Even though applied widely in industry, this method is not sufficiently useful in education environment since test cases are theoretically impossible to secure the absence of possible flaws in programs. In addition, it hardly helps beginning students to trace back to their roots of errors.

In this paper, we suggest a tutoring system known as iiOSProTrain (interactive intelligent Online System for Programming Training) for teaching students programming. For verifying students' programming works and help them correct their own solutions, we combine two recently emerging formal methods for

doi:10.4304/jait.3.1.10-20

imperative program verification, which are theorem proving [13] and model checking [14]. As compared with similar systems, iiOSProTrain possesses the following advantages. Firstly, it can completely confirm the theoretical correctness of solutions submitted by students, due to strong mathematical foundation of theorem proving. It is an important feature for a tutoring system. Being beginners in programming, students would require absolute confirmation on their own works. In some tutoring systems, as reviewed in Section II.1, such confirmation can only be given when students develop their programs in a "templated" format. By contrast, iiOSProTrain enables students conduct their works with full flexibility and creativity. Secondly, iiOSTrain can help students trace back effectively to the roots of their errors, if any, by the means of *error-flow*, a high-level conceptual representations of program flows. The proposed error-flows concepts can overcome the traditional over-technical counter-examples generated by model checking, which would be very hard for novice students (and even professional programmers) to follow.

The rest of the paper is organized as follows. Section II presents some related works on programming tutoring systems and formal methods. In Section III, we discuss the general architecture of iiOSProTrain. Section IV depicts in details the operational mechanism of iiOSProTrain. Section V presents the current status of our system. Finally, Section VI concludes the paper with some future directions pointed out.

II. RELATED WORKS

A. Tutoring Systems for Programming Training

Recently, the advancement of Web-based tutoring systems has prompted some common approaches which are applied widely in various domains such as using visual effects or collaborative manner for coaching the learners. Those techniques are also adopted in some systems specifically intended for programming training. For instance, the educational tool Alice [15] renders learners with a convenient environment to observe theoretical issues through animations illustrating programming concepts. In another attempt, Jeco [16] proposes a mechanism that enables users to finds some possible solutions or hints for their problems from the others works. However, those typical approaches of traditional tutoring system are still not sufficient to cover the basic desire requested by most novice students, which includes the evaluation of their own programs and hints to improve them, or at least to fix the errors occurring if any.

Due to the importance of acquisition of programming skills for computer science students, many attempts has been devoted to help novice students learn to program properly. Early research only focused on syntax issues and had not addressed the problem solving skills significantly required for computer students [17].

Another remarkable approach is adopting software metrics such as Halstead metrics [18] or McCabe complexity [19] to measure the quality of the programs submitted by students [10][20]. This approach is proven useful for marking student's works reasonably and detecting plagiarism effectively. However, in order to improve their program writing skills, students would require more detail feedbacks.

Recently, static analysis techniques and dynamic testing have increasingly been considered adopted for the developments of programming tutoring systems. Static analysis is helpful to check coding style, correctness and efficiency of the algorithms employed in students' works. Notable systems using this method include ASSYST [21], CAP [22] and Expresso [23]. However, this approach is not ideal when there are bugs existing in the submitted programs, since it lacks a mechanism to locate inputs that incur logical errors and generate corresponding counter-examples.

On the other hand, dynamic analysis is also interestingly considered in teaching programming for students. In some online educational systems like TRY [24], BOSS [25] and CourseMaster [26], dynamic analysis is mainly used to automatically mark students' programs. For the sake of convenient communication and interaction, some GUI-enhanced systems are introduced [27], [28]. Recently, APOGEE [29] and AWAT [30] are reported as Web-based systems fully supporting automatic grading of student programs.

To facilitate automatic grading, a typical testing scenario deployed in those systems is as follows. A suit of test cases is first provided to the systems. Then, the systems automatically test students programs based on the test cases and return to students failed test cases. This method works excellently for marking purpose. However, students would require more explanations on the failed test cases to observe the cause of their errors.

Environment for Learning to Program (ELP) [31] is a remarkable system that combines static analysis and dynamic analysis for teaching students Java programming in "fill-in-the-gap" manner. In this system, static analysis is adopted to analyze and compare students' works with predefined solutions [32]. In case of error detected, typical testing approaches of black-box and white-box testing will be applied accordingly [33]. The ELP system thus combines the advantages of both static and dynamic analysis approaches. However, the "fill-in-the-gap" manners applied in this system significantly reduce the flexibility and creativity of students when they are forced to think and infer solutions in a "templated" style.

B. Formal Methods for Programs Verification

B.1. Theorem Proving

The approach of using theorem proving can completely confirm the correctness of a program over some predefined requirements. It consists of two major steps: (i) using logic to describe problem *axiomatically*; and (ii) once again using logic to prove the program correctness.

Axiomatic Problem Description

Axiomatic semantic is closely based on Hoare logic [34], which gives axioms and inference rules for

imperative programming languages. The central feature of Hoare logic is Hoare triple, which is given if the form of {*P*} *C* {*Q*}; where *P* and *Q* are 2 predicates called *precondition* and *postcondition* respectively and *C* is a piece of code called *command*. The semantic of the triple is that if *P* holds then *C* establishes *Q*.

Thus, a Hoare triple is very useful to represent the semantics of imperative statements of programming languages. For example the semantic of the increment operator in C/C++ can be represented as {x==A;} x++; {x==A+1;}. The semantic conveyed is that if the value of variable *x* is *A* before the assignment, the statement x++ will make the value of x become A+1.

With the capability of presenting statement semantics formally, the Hoare logic therefore offers a powerful means to describe a programming problem in a machine-understandable way which allows automatic proof of a program/function correctness. In order to do that, the user first defines a problem as a triple of three components: (i) a natural description; (ii) a precondition; and (iii) postcondition. For example, the problem of finding the maximum between two integers is given as follows:

(P1):
Description: identify the maximum value between two variables *x* and *y*
Precondition: True[1]
Postcondition: {(x>y → result == x) ∨ (result == y)}

Similarly, other problems can be defined in an axiomatic-based manner, as depicted in problem (P2) below.

(P2):
Description: Calculate the absolute value of an integer given as a variable *x*
Precondition: x == A
Postcondition: {(A>=0 → result == A) ∨ (result ==-A)}

Correctness Proving

Once defined axiomatically, a program can be proven in terms of its correctness in a logic-based manner. It is carried out, by the means of axiomatic processing using Hoare logic. We say that a program *C* is correct given precondition *P* and postcondition *Q* if *C* can logically imply *Q* from *P*.

For example, with the problem (P1) defined above, we consider a program (C1) conducted as follows:

(C1): if (x > y) return x; else return y;

According to Hoare logic, in order to prove the correctness of (P1) given (P1), we need to prove

{true} if (x > y) return x else return y {(x>y → result == x) ∨ (result == y)}

After some logic processing, we have the following predicate produced

{x > y} return = x {(x>y → result == x) ∨ (result == y)) ∧ {y >= x} result = y {(x>y → result == x) ∨ (result == y)}

In the logic world, the last predicate is said *valid* (i.e. always holds), hence the correctness of (C1) as a solution for (P1) is claimed. However, this approach experiences difficulty when attempting to "explain" for end-users why a program is incorrect if it is the case.

Theorem Provers

In order to automate the proving process, There are many theorem provers introduced, notably including Z3 [35], Simplify [36], Coq [37], Isabelle [38], Alt-Ergo [39] and Redlog [40]. Some systems, such as Frama-C [41], Why [42] and HIP [43] have been developed to deploy multi-provers in order to make full use of their combined proving capabilities. Based on the mathematical foundation, those provers, once successfully resolving problems, can completely confirm the correctness of the verified systems and produce certified proofs. However, due to the theoretical complexity of logic-based reasoning, those provers suffer major obstacles when dealing with loop-based programs as they need to infer the invariants of the loops, which is a tough task currently far from being completely solved [44].

B.2. Model Checking

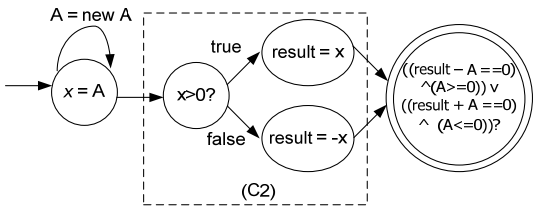


Fig. 1. A formal model of an imperative program.

Model checking is another automatic verification technique in which the system/program to be verified is formalized as a mathematical model, what can be presented as special graph known as *automaton*.

For example, with the problem (P2), a model can be formed as depicted in Figure 1 to check whether the following program (C2) is correct or not.

(C2): if (x >=0) return x; else return –x;

As seen in Figure 1, in order to verify (C2), the model first assigns a certain value of *A* to *x*. Then, after confirming the correct output for this input value, *A* is then reset by a new value and the process is repeated until all of possible values of *A* are completely checked. If there is no wrong output detected during the checking process, the correctness of (C2) is then secured.

[1] *True* means no specific precondition required

To efficiently automate the verification process, the model checker will *unwind* the original model into a finite tree. Thus, to ensure that the number of the nodes in the newly generated tree is finite, a concrete range of possible input values must be provided. For example, in (C2), if *x* is declared as a variable of type *int*, the corresponding range for *A* is [-32768..32767]. Accordingly, the graph-based model in Figure 1 will be unwinded as the tree given in Figure 2.

Hence, to confirm a program/function correctness, the model checking suffers from the state explosion problem as the checker must perform the exhausting search for all of possible values of input parameters. However, in case of error encountered, the search process may terminate fast. Is this problem, the state space for the two input variable x and *y* should be around 65 000 × 65 000 ≈ 4 billion initial states. However, in case of error occurring as exampled, the model checker can generate the counterexample after few first iterations with *x* = -32767 and *y* = -32768. How to "guide" the model quickly converge to the error is still one of the research challenge today.

Nowadays, model checking techniques have been applied widely for verifying hardware designs and protocols since they can check if the system operates as desired or not without actually running the system [45]. Recently, with developed trend of employing formal methods in software engineering, the usage of model checking for software verification is increasingly interesting. Some notable model checkers include Spin [46], Java Pathfinder (JPF) [47], NuSMV [48] and PAT [49].

III. OVERALL ARCHITECTURE

The overall architecture of iiOSProTrain is given in Figure 3. There are three actors working in this system: *Teacher*, *Learner* and *Coordinator*. First, Teacher will provide programming problems, which are presented *axiomatically* in module Problem Description. Then, when Learner visits the system, he can try to solve these problems. The works submitted by Learner are then verified by some theorem provers embedded in the Static Verification module. If an error is detected, then the Dynamic Verification module will generate some helpful counter-examples by the means of model checking. The stochastic information of system, such as common errors or behaviors of active learners is then stored and analyzed in the Analysis Module, which can help the Coordinator to assess the performance of the whole course.

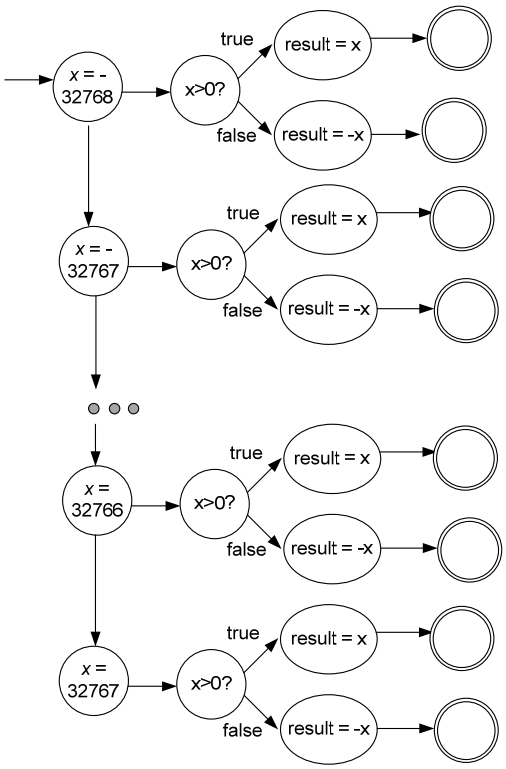


Fig. 2. The unwinded tree generated from the model given in Figure 1.

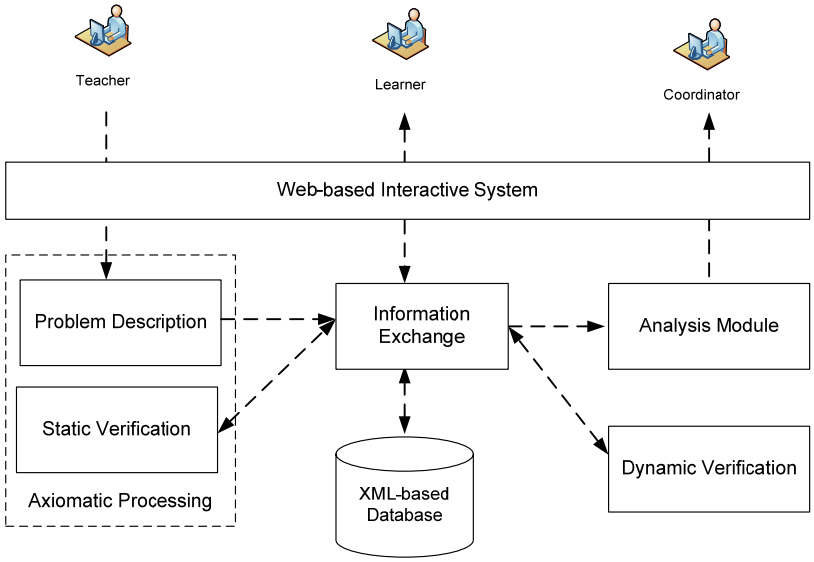


Fig. 3. The proposed verification framework.

For convenient exchange of data between various modules in the framework, the exchanged data in our framework is stored in a XML-based database and then flowed by the Information Exchange module.

Thus, in order to render useful feedback for the programming learners, we combine the two approaches in our system: theorem-proving for axiomatic processing and model checking for counter-example for generation. They will be discussed in more details in the following sections.

IV. OPERATIONAL MECHANISM

A. Problem Description

In iiOSProTrain system, teachers will not provide problem descriptions as in traditional ways. That is, apart from the problem descriptions given in natural language, teachers will also supply the preconditions and postconditions for the system, as depicted in Figure 4. Those preconditions and postconditions will be hidden from students when they attempt to solve the problem, as illustrated in Figure 5.

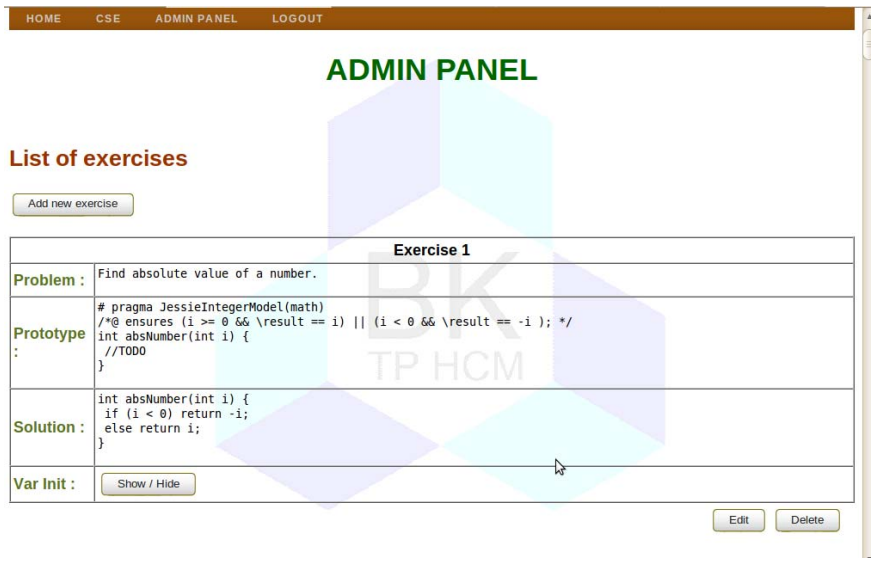


Fig. 4. Teacher describing problems in iiOSProTrain.

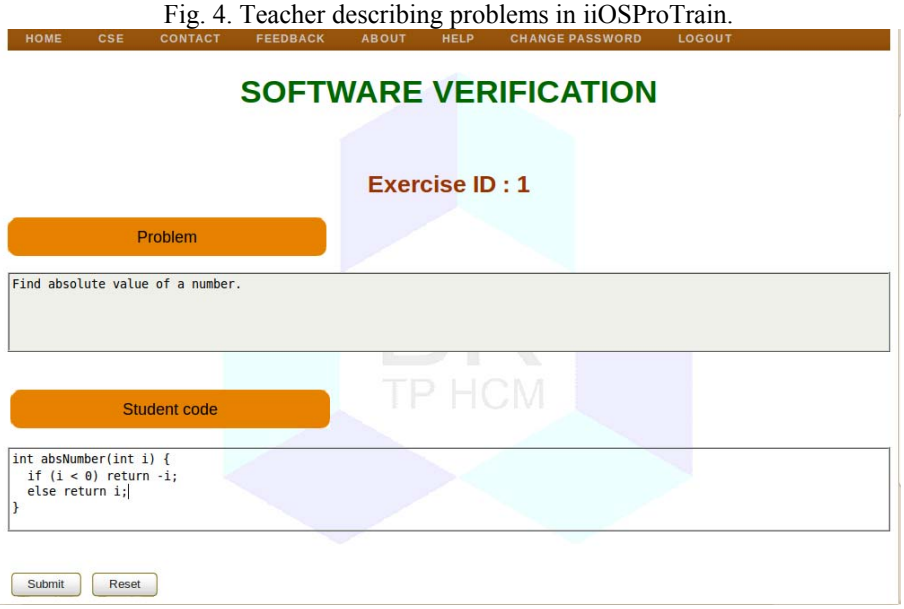


Fig. 5. Learner attempting to solve problems.

B. Static Verification

After a learner submitted a solution for a given problem in iiOSProTrain, the submitted program will be augmented accordingly by the preconditions and postconditions of the problem. Then, the augmented program will be first verified in Static Verification module. As described in Figure 6, Static Verification module performs two main tasks, namely Static Analysis and Correctness Proving. Static Analysis first translates the original source code with formal specification to axiomatic description. In our system, the preconditions and postconditions are written in a dedicated language such as ANSI/ISO C Specification Language (ACSL) [50] which is a behavioral specification language for C program. Axiomatic description is performed in C Intermediate Language (CIL) [51] . Then it is translated to appropriate verification conditions which are used later

for Correctness Proving. In order to be compatible with multiple provers employed, these conditions are stored in an Abstract Syntax Tree (AST) and each prover will parse them to its own supported language if necessary. Based on verification conditions, provers will verify and give the result to programmer. If an error is detected, the program will be passed to Dynamic Verification for generating counter-examples.

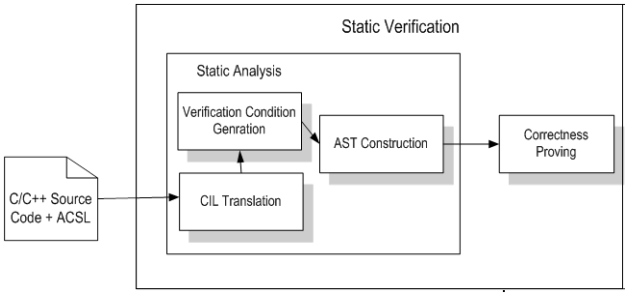


Fig. 6. The Static Verification module.

For instance, Listing 1 depicts three C programs to be verified with pre/post conditions annotated in their comments. In Listing 1(a), we define the problem of finding the absolute value of a given integer with the correct solution submitted to iiOSProTrain. In Listing 1(b), the program submitted to iiOSProTrain is a solution of finding the maximal number between two given integers, which is logically wrong as the programmer accidentally made a mistake in the last statement. For loop-based program, ACSL annotations are more complicated. For instance, the program in Listing 1(c) aims at finding an identified element in the given array. The requires-clause of this program implies that the lower bound and upper bound of the array index must be 0 and n-1 respectively. The ensures-clause is also used to specify the expected returned value of program. Moreover, we define invariants right before each loop statement. Those invariants are essentially required by all provers to verify the loops. In this example, the loop body has an easily observed mistake.

```
   // Find the absolute value of an integer.
   /*@ ensures (i >= 0 && \result == i) || (i < 0
&& \result == -i ); */
   1: int AbsNumber(int i) {
   2:     if (i < 0) return -i;
   3:     else return i;
   4: }
```

(a) Correct solution of a given problem.

```
   // Find the max number between two integers.
   /*@ ensures \result >= x && \result >= y &&
\forall int z; z >= x && z >= y ==> z >=
\result;
   */
   1: int MaxNumber(int x, int y) {
   2:     if (x > y) return y;
   3:     else return y;
   4: }
```

(b) Wrong solution of a given problem.

```
   // In the given array, find the element whose
value equals v and return its first index.
   /*@
   requires \valid_range (t, 0, n-1);
   ensures
      (0 <= \result < n ==> t[\result] == v) &&
      (\result == n ==> \forall int i; 0 <= i < n
==> t[i] != v) ; */
   1: int SearchArray(int t[], int n, int v) {
   2:     int j = 0;
    //invanriant n-j
   3:     while (j < n) {
   4:         if (t[j] == v) break;
   5:         j = j+2;
   6:     }
   7:     return j;
   8: }
```

(c) Loop-based program with an unproved solution.

Listing 1. C codes augmented with formal specifications.

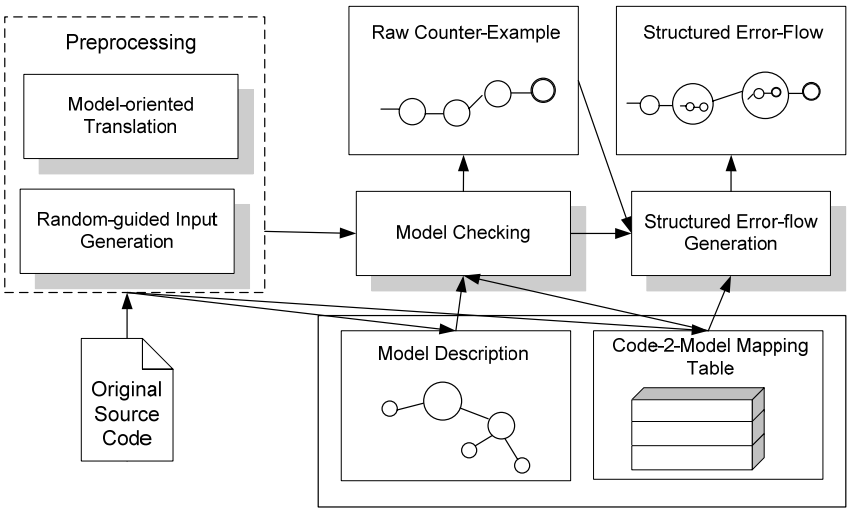


Fig. 7. The Dynamic Verification module.

After affirming that the input programs have no syntax error, the system will then perform static analysis and transfer these programs to the provers to prove if they satisfy the pre-defined requirements or not. Based on the static analysis results, the provers produce verification conditions accordingly. For example, with the program

given in Listing 1(a), there are 2 verification conditions generated, corresponding to 2 cases of $i >= 0$ and $i < 0$. When verified, all conditions are passed by the integrated provers hence the correctness of the program is confirmed[2]. In Listing 1(b), the source code yields six conditions to be verified, among them one is failed when verified. The failure case will then be moved to Dynamic Verification module to be further processed.

By means of axiomatic processing, a program correctness can be easily proven in a formal manner if its pre-conditions, post-conditions and invariants are well-written. However, in case of incorrectness, things become far complicated, especially for loop-based programs. For example, in Listing 2(c), all of our tested provers did not recognize the mistake in the last increasing statement in the loop body. It is not to mention that to require users to specify the loop invariants is also sometimes very troublesome.

C. Dynamic Verification

After the phase of static verification, dynamic verification will be invoked when necessary. As presented in Figure 7, the Dynamic Verification module consists of three major tasks including *Preprocessing*, *Model Checking*, *Structured Error-Flow Generation*.

Preprocessing consists of two subtasks, namely *Model-oriented Translation*, which basically translates the original source code into a formalism of a model checker description; and *Random-Guided Input Generation*, which generates a reduced input space meaningfully in a random–guided manner. In addition, this step will also generate a mapping table between elements of the targeted model checker with those of the original program. This mapping table, known as *Code-2-Model Mapping Table*, will be then used later to retrieve the error-flow in case an error is detected.

Meanwhile, *Model Checking* performs the normal tasks of a model checker: verifying the model against a property, and producing a model-based counter example, known as *raw counter-example*, when recognizing some erroneous problems.

Finally, based on the raw counter-example and the Code-2-Model Mapping Table, *Structured Error-Flow Generation* generates a structured error-flow represented in the original language of the verified program.

C.1. Model-oriented Translation

In dynamic verification, we heavily rely on the model checking technique to verify the program and generate counter-example. To facilitate users' interactions in terms of model description, each model checker usually defines a model description language. Instead of directly stating the internal states of the intended model, which are always complex and highly technical, users can define and modify the model by the means of the corresponding human-understandable description language. Since a model checker can be considered as an "extended automaton", its description language is likely an imperative-like structured language. For instance, in the model checking tool Spin, the description language is Promela [52].

Hence, in this step, we firstly translate the original source code into a form of a model description language of a model checker. For example, in Listing 2 is an example of the C program in Listing 1(b) being translated into Promela for deployed in Spin.

```
proctype max(int x; int y; chan c)
{
            if
            :: x > y ->
              c ! y;
            :: else ->
              c ! y;
            fi;
}

proctype init()
{
  x = 0; y = 0;
  do
  :: x=x+1;
  :: x=x-1;
  :: y=y+1;
  :: y=y-1;
  :: break;
  od;
}
```

Listing 2. Representing a C program in Promela for Spin.

C.2. Random-Guided Input Translation

In this step, we have the model checker verify the translated model. Especially, we developed some tactics of random-guided generation of input values. Thus, it helps to reduce the input space, therefore making the model verification practical for non-trivial real cases. In order to perform random-guided basis, our strategy is to divide the input space into subdomains based on some heuristic rules, some of which are illustrated in Table 1 [53]. For example, Rule 1 states that when encountering a statement of $x > a$, the system will generate three values of x for testing as: $\{a+\epsilon$, RANDOM, MAX$\}$ where ϵ indicating a very small value, MAX the possible maximal value of x and RANDOM a random value between $a+\epsilon$ and MAX.

C.3. Structured Error-flow Generation

For the sake of convenience when learners followthe traces of their errors, in this research we introduce a higher conceptual level of error representation, known as *structured error-flow* [54]. Basically, a structured error-flow is a nested graph-based structure reflecting the execution path performed by a program with a certain input. In order to generate an error-flow, first we analyze the raw counter-example returned by the employed model checker when recognizing a bug. Generally, this raw counter-example is over-specific and thus very hard to be understood and followed by ordinary users. A raw counter-example is illustrated in Figure 8.

[2] For the sake of combined proving capability, there are many provers concurrently adopted in iiOSProTrain, but only one prover confirming the program correctness is sufficient.

Table 1. Heuristic rules for guided-random input generation.

Rule	Statement	Subdomain	Suggested Sample
Rule 1	$x>a$	$(a, +\infty)$	$a+\epsilon$, RANDOM, MAX
Rule 2	$x>=a$	$[a, +\infty)$	a, RANDOM, MAX
Rule 3	$x<a$	$(-\infty, a)$	MIN, RANDOM, $a-\epsilon$
Rule 4	$x<=a$	$(-\infty, a]$	MIN, RANDOM, a
Rule 5	$x==a$	{a}	a
Rule 6	if E …	M_1=Subset(E)	
Rule 7	… elseif E	M_i=Subset(E)$\cap(\neg M_{i-1})\cap\ldots\cap(\neg M_1)$	
Rule 8	… else E	$M_i=(\neg M_{i-1})\cap(\neg M_{i-2})\cap\ldots\cap(\neg M_1)$	

```
2:    proc  0 (:init:) line  25 "verify.spin" (state 7)      [goto :b0]
3:    proc  0 (:init:) line  23 "verify.spin" (state 13)     [(run m_find_max(m1_x,m1_y,c)
4:    proc  1 (m_find_max) line   3 "verify.spin" (state 1)  [(1)]
5:    proc  1 (m_find_max) line   6 "verify.spin" (state 6)  [else]
6:    proc  1 (m_find_max) line   9 "verify.spin" (state 5)  [m1_max = 1]
7:    proc  1 (m_find_max) line  11 "verify.spin" (state 8)  [c!m1_max]
8:    proc  1 (m_find_max) line  12 "verify.spin" (state 10) [(1)]
9:    proc  1 (m_find_max) line   0 "verify.spin" (state 0)  [-end-]
10:   proc  0 (:init:) line  24 "verify.spin" (state 14)     [c?obtainedResult]
```

Fig. 8. A raw counter-example generated by the model checking tool SPIN.

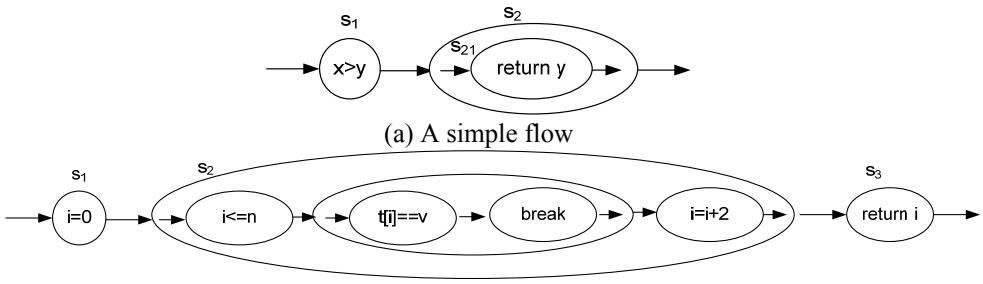
This raw counter-example just simply gives us all of ordered internal states visited by the model checker to achieve the error. We then reply on information stored in the Code-2-Table mapping generated when translating the original program into model description to convert the raw counter-example to a more comprenhensible form.

```
element_type = IF_STRUCTURE
code_line_begin = 2;
model_line_begin = 14;
code_line_end = 3;
model_line_end = 23;
```

Listing 3. Excerpt of recorded information in Code-2-Model Mapping Table

Basically, Code-2-Table is a table-based mapping recording the corresponding information between each program elements (e.g. control structures, loop conditions, etc.) with their counterparts in the model description. The information recorded is just simply the kind of mapped element, the begin and end positions of the elements, in terms of line numbers, in both original program and model description. For instance, the *if* structure in Listing 1(a), when converted into model checking language, is recorded as an entry in the mapping table as depicted in Listing 3.

Thus, information stored in the Code-2-Table mapping can be used to retrieve the original statements corresponding to a raw counter-example. Based on the structural analysis of the retrieved statements, we produce the corresponding final structured error-flow.



(a) A simple flow

(b) A complicated flow

Fig 9. Graph-based representations of structured error-flows

For example, in Figure 9(a) is the structured error-flow generated when testing the program in Listing 1(b) with the input of x and y being 1 and -1 respectively. With these given input, the condition of if-statement is verified and then the return statement is executed. Note that in Figure 9(a), node s_{21} representing the return statement is a subnode inside node s_2, corresponding to the fact that this return statement is inside a loop body. Information in this error-flow can be presented to user in a meaningful manner as depicted in Figure 10.

```
Input:  x = 1; y = -1
Obtained Result: = -1
Desired Result: = 1
Line 3: If statement
        Line 3: [x > y] = true
        Line 4: return y;
```

Fig. 10. Error-flow generated for the program given in Listing 1(b).

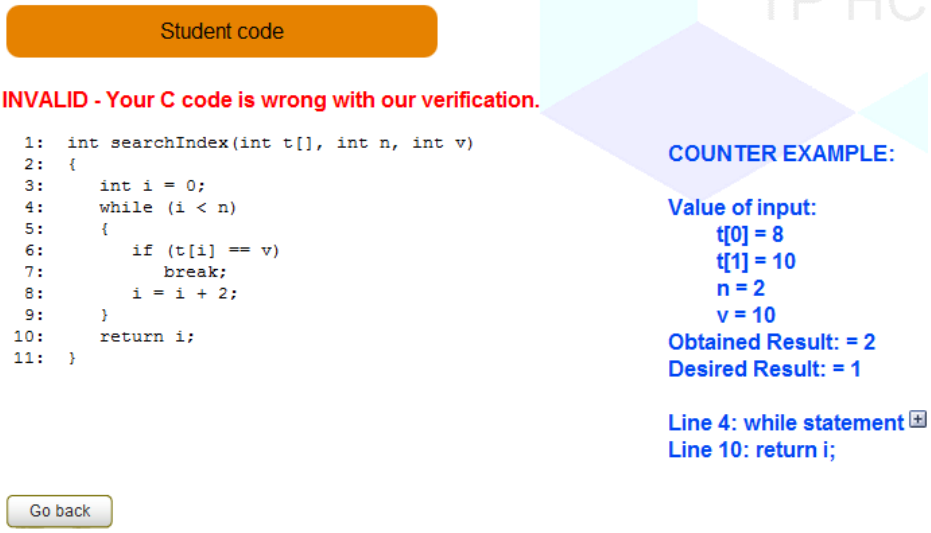


Fig. 11. A structured error-flow presented in iiOSProTrain

The introduced iiOSProTrain tool is being implemented as a Web-based system at Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Vietnam[3]. In the implemented system, we have adopted numerous provers for static verification, including Z3, Simplify and Alt-Ergo. The provers are combined using Frama-C platform with Jessie plugin [55] for static verification. Meanwhile, Spin, a well-known model checker, is adopted for dynamic verification.

The system is intended to improve education performance in the course of Programming Methodologies. The aim of this course is to coach first-year students with the fundamental programming techniques. In iiOSProTrain, we have handled exercises involving if-clause and arithmetic types, including integer and floating-point. Pointer, loop-statement and array are also considered. These exercises are also given in tutorial classes. Therefore, students can verify their own works without much involvements from the tutors. In the first semester that iiOSProTrain is pioneered[4], we recruited a group of 25 voluntary first year students to help test the system. From the Spring semester of 2009, iiOSProTrain is officially launched for all students.

We have receive some positive feedbacks from students as the system can help them to verify their own solutions for the given programming exercises effectively by the means of structured error-flows. Figure 11 illustrates a screenshot of the system generating error-flow for user. In this case, the generated error-flow is quite complicated, as given in Figure 9(b). In Figure 11, there is a '+' symbol appearing at Line 4 notification, which is corresponding to node s_2 in Figure 9(b). When clicking on this symbol, users can continue exploring the statements executions corresponding to the nested subnodes of s_2.

When using this tool to verify students' works, we categorized the verified problems into four types as indicated in Table 2. Type 1 includes the problems involving integer type, linear arithmetic and some simple functions on array of integers. Both static and dynamic verifications worked well in this case. However, static verification was not able to prove the functions in Type 2 including non-linear arithmetic and loop-based programs, which dynamic verification was able to handle. Type 3 contains some simple functions involving floating-point number or recursive algorithm. Static verification performed well in this case, but dynamic verification could not solve them. Our system still experiences problems with functions of Type 4 which involves pointer or complex structured types.

Table 2. Verification results on students work (√ indicates successful verification, whereas ∗ failed cases).

Problems	Static	Dynamic	Combination
Type 1	√	√	√
Type 2	∗	√	√
Type 3	√	∗	√
Type 4	∗	∗	∗

V. CURRENT STATUS

VI. CONCLUSION AND FUTURE WORK

In this paper we introduced an intelligent tutoring systems for teaching programming especially targeting novice students. This kind of learners would always desire prompt and accurate feedback as well as detailed explanation on their works. To fulfill this requirement, we rely on some formal methods popularly considered for software verification in research communities. Thus, our system, known as iiOSProTrain, can render learners absolute confirmation on the correctness of their

[3] One can visit this system as http://elearning.cse.hcmut.edu.vn/provegroup/, username: provegroup, pass: 123456789

[4] In the Spring semester of 2008

submitted programs. In the other case, if there is logic errors detected in student works, a detailed explanation will be returned by the means of structured error-flow, a high-level conceptual representation of execution path. In addition, using formal methods help us avoid the risk of actually running the students code, with may be harmful for the system. Therefore, students are not restricted in any extent when conducting their solutions and freely work with full flexibility and creativity.

Currently, iiOSProTrain is used for teaching first year students and proved useful when able to assist the students working on their weekly programming exercises. Especially, the representation of structured error-flow is very effective to trace back error on loop-based programs, which is always a challenging issue for beginning learners and also a tough research topic on software verification. In the future, our system will also be developed to serve training courses with higher levels of programming like Data Structures or Parallel Programmings. To achieve this vision, it is needed to integrate more advanced techniques in formal methods, such as *separation logic* [56] to deal with pointer-based type and *partial order reduction* [57] with concurrent programs. They are also the directions we will pursue in the near future.

ACKNOWLEDGMENT

This work is part of the Higher Education Project 2 project (supported by World Bank and Hochiminh – Vietnam National University).

REFERENCES

[1] The Joint Task Force: Computing Curricula 2001, Computer Science, ACM, 2001.

[2] Al-Hammadi A.H., Ahmed R.E, and Zualkernan I., Adoption of E-Learning Technology for Anti-Money Laundering Training, *Journal of Advances in Information Technology*, Vol. 1, No. 2, 2010.

[3] Tagga, A. *Modern Sociology*. Lahore: Tagga & Sons Publishers, 2007.

[4] H. Dobler, R. Ramler and K. Wolfmaier. "A Study of Tool Support for the Evaluation of Programming Exercises", *Computer Aided Systems Theory – EUROCAST 2007*, Springer, 2007.

[5] Affleck, G. and Smith, T, "Identifying a need for web-based course support". *Proc. Conference of the Australasian Society for Computers in Learning in Tertiary Education*, Brisbane, Australia, Online, 1999

[6] M. Jazayeri. "The Education of a Software Engineer". *Proceedings of 19th IEEE Conference on Automated Software Engineering*, Linz, Austria, 2004.

[7] Ben-Ari, M. "Constructivism in Computer Science Education". Journal *of Computers in Mathematics & Science Teaching* 20(1): 2001, 24-73.

[8] Sanders, D. and Hartman, J. "Assessing the quality of programs": *A topic for the CS2 course*. Proc, 1987.

[9] Norcio, A. F. "Human Memory Processes for Comprehending Computer Programs". Proc. IEEE Systems, Man and Cybernetics Society, Cambridge, Massachusetts, 1980, 974-977, IEEE.

[10] Mengel, S. And Yerramilli, V. "A Case Study Of The Static Analysis Of the Quality Of Novice Student Programs". Proc. *Thirtieth SIGCSE technical symposium on Computer science education*, New Orleans, Louisiana, United States, 13:78-82, 1999.

[11] K.E. Wiegers. *Peer Reviews in Software*, Addison Wesley, London, UK, 2002.

[12] A. Spillner, T. Linz, H. Schaefer. *Software Testing Foundations*, dpunkt, 2006.

[13] D. A. Duffy, "Principles of Automated Theorem Proving", John Wiley & Sons, 1991.

[14] E. M. Clarke and E. A. Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". *Logic of Programs*: 52-71, 1981

[15] Carnegie Mellon University. *Alice Education Software*, available at http://www.cmu.edu/corporate/news/2007/features/alice.shtml

[16] A. Moreno, N. Myller and E. Sutinen. "JECO, a Collaborative Learning Tool for Programming", *Proceedings of 2004 IEEE Symposium on Visual Languages and Human Centric Computing*: 261-263, 2004.

[17] Deek, F. and Mchugh, J. "A survey and critical analysis of Tools for Learning Programming". *Journal of Computer Science Education*, 8(2): 130-178, 1998.

[18] Halstead, M. H. "Elements of software science", Elsevier, New York, 1977.

[19] Mccabe, T. J. "A Complexity Measure". IEEE Transactions on Software Engineering, 2(4): 308-320, 1976.

[20] Leach, R. J. "Using metrics to evaluate student programs". *ACM SIGCSE Bulletin*, 27(2): 41-43, 1995.

[21] Jackson, D. "A software system for grading student computer programs". *Proc. the twenty-eighth SIGCSE technical symposium on Computer science education*, San Jose, California, United States 28: 335-339, ACM Press, 1997.

[22] Schorsch, T. "CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors". Proc. *The twenty-sixth SIGCSE technical.* 1995.

[23] Hristova, M., Misra, A., Rutter, M. and Mercuri, R. "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students". *Proc.the 34th SIGCSE technical symposium on Computer science education*, Reno, Nevada, USA, 34:153-156, ACM Press. 2003.

[24] Reek, K. A. "The TRY system -or- how to avoid testing student programs". *Proc. The twentieth SIGCSE technical symposium on Computer science education*, Louisville, Kentucky, United States, 21:112-116, ACM Press New York, NY, USA, 1989.

[25] Joy, M., Griffiths, N., and Boyatt, R. "The boss online submission and assessment system". *J. Educ. Resour. Comput.* 5, 3 (Sep. 2005), 2.

[26] CourseMaster: School of Computer Science & IT, The University of Nottingham, UK. http://www.cs.nott.ac.uk/CourseMaster/cm_com/index.html . Accessed 2002

[27] Sun, Y. and Jones, E. L. "Specification-driven automated testing of GUI-based Java programs". In *Proceedings of the 42nd Annual Southeast Regional Conference* (Huntsville, Alabama, April 02 - 03, 2004). ACM-SE 42. ACM, New York, NY, 140-145.

[28] Man Yu Feng; Mcallister, A.; "A Tool for Automated GUI Program Grading," *Frontiers in Education Conference, 36th Annual* , vol., no., pp.7-12, 27-31 Oct. 2006

[29] Fu, X., Peltsverger, B., Qian, K., Tao, L., and Liu, J. "APOGEE: automated project grading and instant feedback system for web based computing". *SIGCSE Bull.* 40, 1 (Feb. 2008), 77-81

[30] Sztipanovits, M., Qian, K., and Fu, X. "The automated web application testing (AWAT) system". In *Proceedings of the 46th Annual Southeast Regional Conference on XX* (Auburn, Alabama, March 28 - 29, 2008). ACM-SE 46. ACM, New York, NY, 88-93

[31] Truong, N., Bancroft, P., and Roe, P. 2003. A web based environment for learning to program. In *Proceedings of the 26th Australasian Computer Science Conference - Volume 16* (Adelaide, Australia). M. J. Oudshoorn, Ed. ACSC, vol. 35. Australian Computer Society, Darlinghurst, Australia, 255-264.

[32] Truong, N., Roe, P., and Bancroft, P. "Static analysis of students' Java programs". In *Proceedings of the Sixth Conference on Australasian Computing Education - Volume 30* (Dunedin, New Zealand). R. Lister and A. Young, Eds. ACM International Conference Proceeding Series, vol. 57. Australian Computer Society, Darlinghurst, Australia, 317-325, 2004.

[33] Truong, N., Roe, P., and Bancroft, P. "Automated feedback for "fill in the gap" programming exercises". In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42* (Newcastle, New South Wales, Australia). A. Young and D. Tolhurst, Eds. ACM International Conference Proceeding Series, vol. 106. Australian Computer Society, Darlinghurst, Australia, 117-126, 2005.

[34] M Huth and M. Ryanl, *Logic In Computer Science: Modeling And Reasoning About Systems*, Cambridge University Press, 1999.

[35] Z3: An Efficient SMT Solver, available at http://research.microsoft.com/en-us/um/redmond/projects/z3/

[36] Detlefs, D., Nelson, G., and Saxe, J. B. "Simplify: a theorem prover for program checking". *J. ACM* 52, 3 (May. 2005), 365-473.

[37] The Coq Proof Assistant, available at http://www.lix.polytechnique.fr/coq/

[38] Isabelle, available at http://www.cl.cam.ac.uk/research/hvg/Isabelle/

[39] The Alt-Ergo Theorem Prover, available at http://alt-ergo.lri.fr/

[40] Redlog, avaliable at http://redlog.dolzmann.de/

[41] Frama-C, available at http://frama-c.com/

[42] Why: a software verification platform, available at: http://why.lri.fr/.

[43] HIP Overview, available at http://loris-7.ddns.comp.nus.edu.sg/~project/hip/index.html

[44] Flanagan, C. and Qadeer, S. "Predicate abstraction for software verification". In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, January 16 - 18, 2002). POPL '02. ACM, New York, NY, 191-202.

[45] Clarke, E.M.; Kurshan, R.P.; "Computer-aided verification," *Spectrum, IEEE* , vol.33, no.6, pp.61-67, Jun 1996.

[46] Spin – Formal Verification, available at http://spinroot.com/

[47] Java PathFinder, available at http://babelfish.arc.nasa.gov/trac/jpf

[48] NuSMV – a new symbolic model checker, available at http://nusmv.fbk.eu/

[49] J. Sun, Y. Liu, J.S. Dong and J. Pang. "PAT: Towards Flexible Verification under Fairness". *In Proceedings of the 21th International Conference on Computer Aided Verification* (CAV 2009), Grenoble, France, June, 2009.

[50] P. Baudin, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy and V. Prevosto. "ACSL: ANSI/ISO C Specification Language", preliminary design (version 1.4). October 2008.

[51] G. C. Necula , S. Mcpeak , S. P. Rahul and W. Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs". *In International Conference on Compiler Construction*, pp. 213-228, 2002.

[52] The PROMELA language, available at http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html

[53] Quan, T.T.; Hoang, D.L.N.; Nguyen, B.T.; Nguyen, A.N.; Tran, Q.D.; Nguyen, P.H.; Bui, T.H.; Do, A.T.; Huynh, L.V.; Doan, N.T.; Huynh, N.T.; Nguyen, T.D.; Nguyen, T.T.; Nguyen, V.H.; 9-11 June 2010 "MAFSE: A Model-Based Framework for Software Verification," *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on* , vol., no., pp.150-156.

[54] Quan T.T., Hoang D.L.N., Nguyen.V.H. and Nguyen P.H. 2010., "Model–based Generation of Structured Error–Flows in Imperative Programs", *In Proceedings of International Conference on Advanced Computing and Applications (ACOMP 2010)*.

[55] JESSIE plugin, available at http://frama-c.cea.fr/jessie.html

[56] Separation Logic: A Logic for Shared Mutable Data Structures John Reynolds. Proceedings of LICS 2002. pp55-74

[57] P. Godefroid, Using partial orders to improve automatic verification methods, Computer-Aided Verification, LNCS 531, pp. 176-185, Springer, 1991.

Dr. Quan Thanh Thanh obtained his BS and PhD in Computer Science from the Ho Chi Minh City University of Technology (HCMUT) and Nanyang Technological University (NTU) in 1998 and 2006 respectively. He is currently Head of Department of Computer Science, Faculty of Computer Science and Engineering at HCMUT. His research interests include Formal Methods, Software Verification and Intelligent Systems.