Accepted for publication. To appear in *IEEE Transactions on Software Engineering.*

Transformation-based Diagnosis of Student Programs for Programming Tutoring Systems

Songwen Xu and Yam San Chee

Abstract: A robust technology that automates the diagnosis of students' programs is essential for programming tutoring systems. Such technology should be able to determine whether programs coded by a student are correct. If a student's program is incorrect, the system should be able to pinpoint errors in the program as well as explain and correct the errors. Due to the difficulty of this problem, no existing system performs this task entirely satisfactorily, and this problem still hampers the development of programming tutoring systems. This paper describes a transformation-based approach to automate the diagnosis of students' programs for programming tutoring systems. Improved control-flow analysis and data-flow analysis are used in program analysis. Automatic diagnosis of student programs is achieved by comparing the student program with a specimen program at the semantic level after both are standardized. The approach was implemented and tested on 525 real student programs for nine different programming tasks. Test results show that the method satisfies the requirements stated above. Compared to other existing approaches to automatic diagnosis of student programs, the approach developed here is more rigorous and safer in identifying student programming errors. It is also simpler to make use of in practice. Only specimen programs are needed for the diagnosis of student programs. The techniques of program standardization and program comparison developed here may also be useful for research in the fields of program understanding and software maintenance.

Index Terms—Automatic program diagnosis, programming tutoring systems, program representation, program transformation, program comparison, program matching

1. Introduction

The automatic diagnosis of students' programs is essential for programming tutoring systems [16]. For example, if we are constructing a web-based online programming tutoring system, in order to apply various pedagogical strategies to realize individualized teaching, such a system must have a technology to check the correctness of the programs submitted by the students. In this context, a program refers to a procedure, a function, or a method.

The purpose of automatic diagnosis of student programs is to identify semantic errors in student programs automatically. A student program may contain two kinds of errors: syntax errors and semantic errors. In the context of programming tutoring systems, the problem of detecting syntax errors in a student program has been solved satisfactorily. The system can invoke a compiler of the programming language in which a student program is written to detect the syntax errors in the student program. It can also deploy the same techniques in the compiler to parse the student program directly to detect syntax errors in the student program. However, the problem of detecting semantic errors in a student program is not so readily solved because there are no equivalent tools, corresponding to a compiler, that can be deployed by a programming tutoring system to detect semantic errors in a student program. Therefore, in this research, the objective of the work is to develop a technology that enables a programming tutoring system to automate the detection of semantic errors in student programs. More specifically, the technology should satisfy the following three requirements. (1) It should be able to determine whether a student program is semantically

correct. (2) It should be able to locate errors in an incorrect program. (3) It should provide explanations and corrections to the errors identified.

The diagnosis of student programs is a difficult problem in the development of programming tutoring systems because many variations may exist in a student program that satisfies the requirements specified in the programming task. These variations are not semantic errors in the student program. We call these variations *semantics-preserving variations*. The challenge in the diagnosis of student programs rests in how we can devise a method to recognize correct student programs, which may contain many semantics-preserving variations, and also correctly identify incorrect student programs, providing the necessary indications and explanations of the errors to students.

The most straightforward approach to the problem is to run the student program to verify whether it produces the proper results expected. In the context of programming tutoring systems, however, this approach has two limitations. First, it is not able to locate errors in incorrect student programs. Second, in some circumstances, it is impossible to obtain the environment required for running a student program.

Research into the automatic diagnosis of student programs can involve many different fields including intelligent programming tutoring systems, automatic program assessment [25], automatic program understanding [3], [23], program representation [19], and program semantics. Due to the difficulty of the problem of automatic diagnosis of student programs,

existing approaches to the automatic diagnosis of student programs are unable to satisfy the above requirements due to various difficulties discussed below.

To date, three main approaches to the automatic diagnosis of student programs have been used. They differ with respect to the ways in which knowledge of correct student programs are requested and handled. The first approach attempts to match a student's program against a specification that is a high-level description of the program's goals [1], [2], [7], [9], [14], [17], [24], [26]. This approach might be referred to as a *source-to-specification* approach. The approach has been applied to functional programming languages such as Lisp [2], [7], and Prolog [17], [24]. It has also been applied to imperative programming languages such as Pascal [1], [14], [26] and C [9]. The *source-to-specification* approach is consistent with the research on program understanding [3], [23]. An essential step in the approach is to understand the student program by matching it with code templates that are attached with corresponding specifications [14], [17], or matching it with specifications called goal structures [1], [2], schemata [7], plans [9], assertions [24], or processes [26]. However, several problems arise when using this approach. First, unlike matching student programs using code templates for functional languages such as Lisp and Prolog [2], [7], [17], [24] or matching student programs in imperative programming languages such as Pascal and C for a limited number of pre-designed programming tasks [14], it is extremely difficult to match *any* programs in *any* programming language, including object-oriented programming languages such as Smalltalk, C++, and Java, because there are too many possible variations permissible in students' programs. This difficulty of program matching affects the ability of systems to successfully recognize student programs using the *source-to-specification* approach. Second,

it is difficult to obtain the goal description of a program by asking the student questions about its desired behavior, following the approach used in [24]. It is also very difficult for instructors to write specifications that will be used to match the student programs. For instructors, the simplest way of describing the solution of a programming task is to write a program for the task rather than the specifications themselves. Third, matching a student's program with a program specification is not a rigorous procedure because the problem does not exist in the realm of provable reasoning. This difficulty in program matching also hampers the research on program understanding.

The second approach to the automatic diagnosis of student programs attempts to extract the formal specifications of both a student program and a reference program and then to perform automatic reasoning upon the extracted formal specifications [21]. This approach is a *specification-to-specification* approach. In *Talus* [21], the Boyer-Moore logic provides the formal specifications of both student programs and reference programs in Lisp. *Talus* also uses the Boyer-Moore logic to automate the reasoning about program semantics. The *specification-to-specification* approach would be an ideal approach if the formal specifications of programs can be derived automatically. However, it is still not possible to produce formal specifications automatically for real-world programs in Pascal, C/C++, Smalltalk, and Java, let alone reason about the formal specifications automatically. Therefore, automated *specification-to-specification* diagnosis is still infeasible at this time. Due to problems associated with each of the above approaches, the two approaches discussed so far are not widely used as practical technologies for automatic diagnosis of student programs in programming tutoring systems.

The third approach attempts to match a student program against a specimen program (also called a model program) stored in the system. This approach can be called a *source-to-source* approach. It differs from the *specification-to-specification* approach in that the latter performs automatic reasoning at the level of formal specifications, whereas the *source-to-source* approach performs automatic reasoning at the level of program dependence graphs. A program dependence graph is a semantic representation for programs, but it is not a formal specification. An early piece of work using this approach is the LAURA system for learning FORTRAN [4]. LAURA uses very primitive program representations and transformations that are not applicable to structured languages and object-oriented languages. The match between the student program and the model program is not carried out rigorously at the semantic level.

Initial advances in program representation and program transformation are reflected in *Recognizer* [23], a prototype demonstrating the feasibility of automatic program recognition using a graph-parsing approach. Programs are represented in program dependence graphs [6]. They are transformed before being matched against pre-stored commonly used programming structures, called *clichés.* However, the program representation and program transformations used are still primitive, and a more rigorous matching technique is required for *cliché* matching. With further advances in the fields of program analysis and compiler design, technologies such as Object-oriented Program Dependence Graph (OPDG) representation [19], program transformation [10], [15], [18], [20], [22], and program comparison [11], [13], [27], have become more mature. Progress along these lines has made

the technology of matching a student's program with a model program more attractive and feasible. Hence, the *source-to-source* approach, which is based on the idea of matching a student's program with a model program, offers fresh promise for investigating the automatic diagnosis of student programs.

In this paper, we describe a method, based on the *source-to-source* approach, to automate the diagnosis of student programs for programming tutoring systems. In this approach, model programs are used as input to the diagnosis of student programs. The automatic diagnosis of students' programming errors is achieved by comparing the student program with a model program after both have been standardized by program transformations. We refer to our approach as *transformation-based diagnosis.*

Transformation-based diagnosis is a general diagnosis technique. It can be used to diagnose programs in object-oriented programming languages such as Smalltalk, C++, and Java as well as programs in non-objected-oriented programming languages such as Pascal and C. In our work, we demonstrate the implementation of transformation-based diagnosis using Smalltalk in *SIPLeS-II,* an automatic program diagnosis system for programming tutoring systems. The system has been tested on 525 student programs for nine different programming tasks. Test results show that the method satisfies the three requirements for the automatic diagnosis of student programs described above.

Although transformation-based diagnosis is not a complete solution to the problem of automatic diagnosis of student programs, it has two main merits when compared to other

existing approaches. (1) It is more rigorous and safer in identifying student programming errors. (2) It is simpler to make use of in practice. Only specimen programs are needed for the diagnosis of student programs. Knowledge bases, which require pre-empirical study of the programming tasks, are not needed.

The main techniques developed in this work are program standardization and semantic-level program matching. These techniques may aid further development of the *source-to-specification* diagnosis approach by enhancing its program matching ability. It may also be useful for research in the fields of program understanding and software maintenance.

Some limitations of the method are: (1) Transformation-based diagnosis does not encompass intentions underlying student programs because it does not encode the knowledge needed to deal with intentional information. However, the absence of such knowledge and associated knowledge base makes the transformation-based approach simpler to deploy in practice. (2) Similar to the other approaches referred to above, transformation-based diagnosis is an *intra-procedural* diagnosis method. It follows that handling variations at the *system* level, such as dealing with different class hierarchies and handling *inter-procedural* variations, are beyond the scope of the approach.

In the following sections of this paper, we first overview the transformation-based diagnosis approach. Then, we describe four aspects of transformation-based diagnosis—program representations, program standardization, program comparison, and error detection—using a running example. Finally we present our test results and discuss them.

2. An overview of transformation-based diagnosis

In transformation-based diagnosis, for a given programming task, different model programs are used to diagnose students' programs. These student programs may use different algorithms. In principle, a correct student program should be recognized as being equivalent to the corresponding model program after both have been standardized using a set of standardization transformation rules. Semantic differences between the two programs identified in the comparison indicate programming errors in the student programs.

2.1 Theoretical foundation of transformation-based diagnosis

In the context of programming tutoring systems, the semantic correctness of a student program is defined as the correctness of *computational semantics* rather than the correctness of *operational semantics.* Two programs have equivalent computational semantics if they give the equivalent running results although the two programs may use different algorithms and thus may have different operational semantics. Two programs have equivalent operational semantics if they use the same algorithm and have the same execution behaviors. The theoretical foundation of transformation-based diagnosis is based on the following assertions in the context of programming tutoring systems.

Assertion1 In transformation-based diagnosis, the semantic correctness of a student program for a programming task can be defined by using a set of model programs, which correspond to a set of algorithms to solve the programming task.

Assertion2 If the semantics of a student program is equivalent to the semantics of one of the model programs, then the student program is semantically correct.

Assertion3 If the semantics of a student program is not equivalent to any of the model programs, then the student program is regarded as semantically incorrect.

Assertion4 Discrepancies between the semantics of a student program and the semantics of a model program that uses the same algorithm as that of the student program reveal semantic errors in the student program and also indicate the necessary corrections for the errors.

Assertion5 AOPDG (Augmented Object-oriented Program Dependence Graph) is a semantic program representation that represents the operational semantics of programs, not the computational semantics. It combines the OPDG representation [19] with the features of static-single-assignment form [20]. Two programs with equivalent AOPDGs have equivalent semantics. Two programs using the same algorithm with equivalent semantics may have different AOPDGs.

Assertion6 A transformation is a *semantics-preserving transformation* if it changes the computational behaviors (i.e., operational semantics) of a program while preserving the computational results (i.e., computational semantics). A sufficiently long sequence of semantics-preserving transformations can standardize all semantically equivalent programs that use the same algorithm into those having the same AOPDG.

Assertion7 Based on Assertion5 and Assertion6, two standardized programs with equivalent AOPDGs have equivalent semantics. Two standardized programs using the same algorithm with equivalent semantics have equivalent AOPDGs.

Assertion8 Based on Assertion2 and Assertion7, if the AOPDG of a standardized student program is equivalent to the AOPDG of one of the standardized model programs using the same algorithm, then, the student program is semantically correct.

Assertion9 Based on Assertion3 and Assertion7, if the AOPDG of a standardized student program is not equivalent to any of the standardized model programs then, the student program is regarded as semantically incorrect.

Assertion10 Based on Assertion3 and Assertion8, discrepancies between the AOPDG of a student program and the AOPDG of a model program that uses the same algorithm as that of the student program reveal semantic errors in the student program and also indicate the necessary corrections to the errors.

Therefore, transformation-based diagnosis can identify the semantic discrepancies between a student program and a model program by matching the AOPDG of the standardized student program against the AOPDG of the standardized model program.

From the above reasoning, it is clear that the following questions need to be answered in transformation-based diagnosis. These questions are all addressed in subsequent parts of the paper.

(1) What is the appropriate program representation for program standardization, and how do we standardize programs?

(2) What is the appropriate program representation for program matching, and how do we match programs at semantic level?

(3) How do we recognize all semantically correct programs by dealing with all kinds of variations that may appear in them?

(4) How do we identify all kinds of errors in incorrect programs and obtain explanations for the errors as well as indicate necessary corrections to the errors?

2.2 Possible semantics-preserving variations in a student program

In programming tutoring systems, a functionally correct program written by a student may have many semantics-preserving variations (SPV) compared to a model program. Based on our empirical study of more than 200 student programs written in various programming languages, we arrived at the following 13 possible types of SPVs.

(1) SPV1—Different algorithms. SP and MP use different algorithms in solving a programming task but their computational outputs are the same. For example, different algorithms can be used to implement sorting.

(2) SPV2—Different source code formats. SP and MP may have differences purely at the source code level, and these can be eliminated when the programs are parsed. For example, more or fewer spaces, comments, etc. are used.

(3) SPV3—Different syntax forms. SP and MP may use different syntax forms in expressing a certain program element. For example, in Smalltalk, a message sequence can be either written in the form of several statements or in the form of cascaded messages in a single statement.

(4) SPV4—Different variable declarations. SP and MP may use different temporary variable declarations. For example, temporary variables may be declared in a method temporary variable declaration or in a block temporary variable declaration.

(5) SPV5—Different algebraic expression forms. SP and MP may use different algebraic expressions forms.

(6) SPV6—Different control structures. SP and MP may use different control structures in expressing the same control, such as branching, iterating, and traversing.

(7) SPV7—Different Boolean expression forms. SP and MP may use different Boolean expression forms.

(8) SPV8—Different temporary variables. SP and MP may use different numbers of temporary variables or block temporary variables.

(9) SPV9—Different redundant statements. SP and MP may contain some dead code or some statements included only for debugging purpose.

(10) SPV10—Different statement orders. SP and MP may have statement placed in different order.

(11) SPV11—Different variable names. SP and MP may use different parameter names, different temporary variable names, or different block temporary variable names.

(12) SPV12—Different program logical structures. SP and MP may have different program logical structures. For example, some statements can be placed either inside a loop or out of a loop.

(13) SPV13—Different statements. SP and MP may contain different statements. Some components in a SP statement may be different from those in a MP statement although the computational results of the two statements are the same.

There could be other semantics-preserving variations. For example, several elements distributed in several SP statements may differ from elements in several MP statements. We classify such multiple-statement variations as SPV1—using different algorithms. In this way, all SPVs between two programs can be classified as one of the13 SPVs listed above.

By applying various techniques—program representation, program standardization, program comparison, and error detection—all SPVs are handled using transformation-based diagnosis.

2.3 An overview of program representation, program standardization, program comparison, and error detection

In transformation-based diagnosis, programs are represented using three forms of representation: (1) source code, (2) Abstract Syntax Tree (AST), and Augmented Object-oriented Program Dependence Graph (AOPDG). The AST representation is amenable to program analysis and program transformations. The AOPDG representation combines the OPDG representation [19] with the features of static-single-assignment form [20]. It is used for the program matching at the semantic level. The merits of the AOPDG representation are: (1) it eliminates many variations that exist at the source code level and AST level, (2) it supports the program-matching algorithm, and (3) it enables the matching algorithm to

recognize semantics-preserving behavioral changes in the student program compared to the model program.

In order to standardize programs, a set of program standardization transformation rules is identified. These rules have been identified after a study of possible variations that may appear in student programs. Two kinds of standardization transformations are used: *basic standardization transformations* and *advanced standardization transformations.*

The program-matching algorithm used in transformation-based diagnosis utilizes the idea that two statements with different operators, different operands, or different controlling predicates will have different behaviors. The vertices in the student AOPDG (i.e. the statements in the student program) and the vertices in the model AOPDG are first classified into different classes in an initial partition according to their behaviors. Next, a stable coarsest refinement of the initial partition is computed using a basic partitioning algorithm. After the matching process, discrepancies that remain between the AOPDGs of the two programs reveal semantic differences between the student program and the model program. Semantics-preserving variations between the two programs can be recognized correctly.

An error detection phase is employed to handle semantics-preserving variations left over from program standardization and program matching. First, textual variations between semantically equivalent statements are identified and eliminated. Second, some semantics-preserving variations are learned as heuristic knowledge by interacting with the user. Third,

the variations that remain in student programs are identified as semantic errors. Explanations and necessary corrections to the errors are also provided.

2.4 The automatic diagnosis procedure

The procedure of transformation-based diagnosis is shown in Fig. 1.

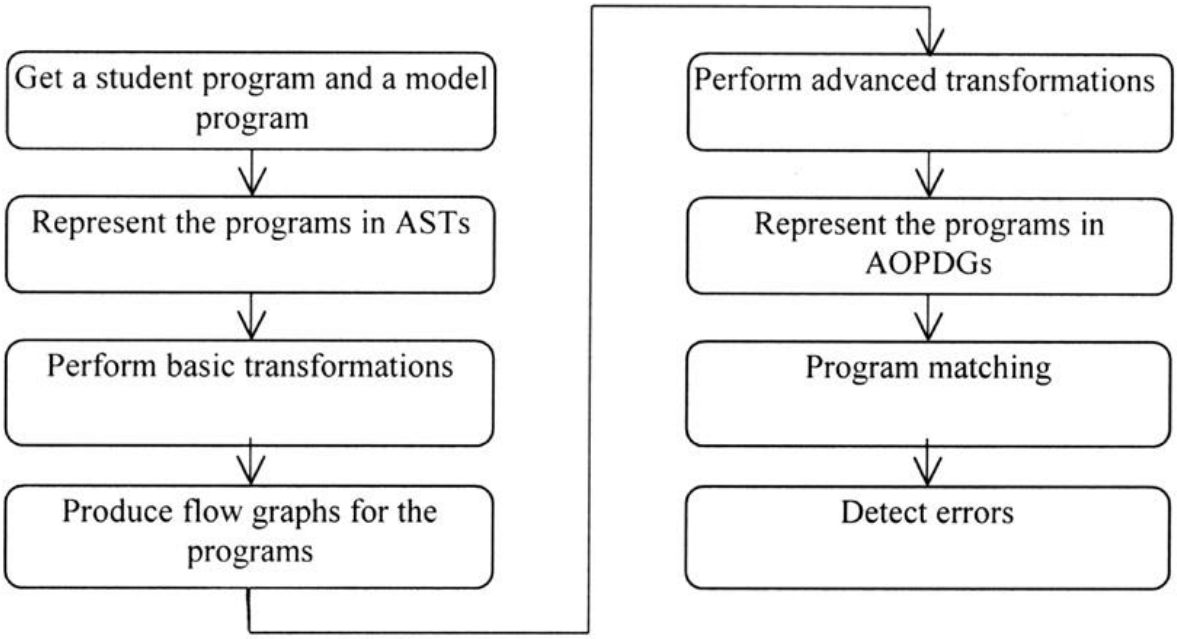


Fig. 1. Procedure of transformation-based diagnosis

Get a student program and a model program: a student program (called SP) and a model program (called MP) are input for diagnosis based on their original source code.

Represent the programs in ASTs: Programs are first processed by a parser, and the parse trees of the two programs are generated. Based on a set of Backus-Naur Forms (BNF) [12] for the language that the programs are written in, the AST representations of the two programs are produced. These representations are called SPTree and MPTree.

Perform basic transformations: Basic transformations that do not require definition-use information (DU information) are performed to standardize SPTree and MPTree.

Produce flow graphs for the programs: In order to produce AOPDGs, flow graphs for the student program and the model program are produced based on SPTree and MPTree. Flow graphs are augmented with extra vertices [27] in order to combine them with features of Static Single Assignment (SSA) form [20] where every use of a variable is only defined by one definition. The DU information for the two programs, which is necessary for advanced transformations, is calculated. This information will also be used later in the generation of AOPDGs.

Perform advanced transformations: The advanced transformations, which require DU information, are performed to standardize SPTree and MPTree further.

Represent the programs in AOPDGs: The AOPDGs for SP and MP are produced based on the flow graphs for the student program and the model program. By control-flow analysis, the system obtains Augmented Object-oriented Control Dependence Subgraphs (AOCDS) of AOPDGs. By data-flow analysis, the system obtains Augmented Object-oriented Data Dependence Subgraphs (AODDS) of the AOPDGs. We call the AOPDGs for SP and MP SPGraph and MPGraph respectively.

Compare the programs: A program matching algorithm compares SPGraph and MPGraph and identifies semantic differences between the student program and the model program. It produces the following outputs: (1) A mapping between statements in SP and their semantics-equivalent statements in MP. We call this the *equivalent map*. (2) A mapping between statements in SP and statements in MP that are semantics-equivalent but textually different from the statements in SP. We called this the *textual difference map.* (3) A set that includes unmatched statements in SP and MP. This set is called the *unmatched set.*

Detect errors: The comparison results are processed further in an error detection step. Briefly, the reasoning and processes used in this step are as follows. The SP statements in the equivalent map and the textual difference map are recognized as correct statements. For every unmatched SP statement, the most similar unmatched MP statement is found, and the differences between the two statements are identified in SP. Among these differences, those that are actually legal variations of SP are learned and eliminated by the system. Unresolved differences are reported as errors in the diagnosis report. However, some SP statements might be semantically unmatched only due to control errors in other statements. Such an unmatched SP statement can textually match the most similar unmatched MP statement with no superficial difference. These SP statements are reported in the diagnosis report as semantic errors because of control structure errors elsewhere.

Details of the four aspects of transformation-based diagnosis—program representation, program standardization, program comparison, and error detection—are described in the following sections with the aid of a running example.

3. A running example

The task description

Define a method called taxiFeeWith: mile isBookingCase: bookingCase. If "bookingCase" is true, a booking fee of $2.00 should be charged, and the price per mile is $3.00; otherwise, no booking fee is charged, and the price per mile is $2.50. The total taxi fee is calculated by mile*price + bookingFee.

An algorithm for the task

```
{{initialize temporary variables of 'bookingFee' and 'price'}
{if bookingCase is true
      {set new values of bookingFee and price}}
{calulate the total taxifee}}
```

Box 1. An algorithm for task taxiFee

A model program (MP)

A model program based on the above algorithm is shown in Box 2, where names for the method head and statements are shown on the left-hand side.

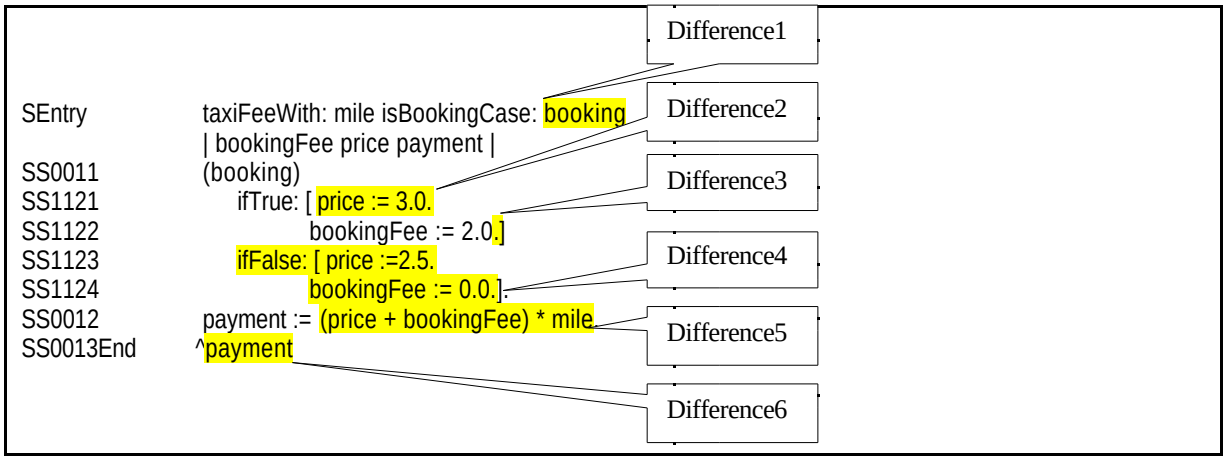
```
MEntry        taxiFeeWith: mile isBookingCase: bookingCase
              | bookingFee price |
MS0011        bookingFee := 0.0.
MS0012        price :=2.5.
MS0013        (bookingCase)
MS1321            ifTrue: [bookingFee := 2.0.
MS1322                    price := 3.0 ].

MS0014End     ^(price *mile + bookingFee)
```

Box 2. A model program for task taxiFee

A student program (SP)

A possible student program is shown in Box 3. Differences between SP and MP are highlighted and discussed later.



Box 3. A student program for task taxiFee

Differences between SP and MP

- Difference1 (SPV 11): SP uses different parameter names in SEntry and SS0011 compared to that used by MP in MEntry and MS0013.
- Difference2 (SPV 10): SP uses different statement orders of SS1121 and S1122 compared to that of MS1321 and MS1322 in MP.
- Difference3 (SPV 2): SP has a different source-code format in SS1122 compared to that in MS1322 of MP. There is an extra "." before the "]" in SS1122.
- Difference4 (SPV 12): SP and MP produce different values of price and bookingFee before SS0011. In MP, price and bookingFee are calculated *outside* of the ifFalse: control structure, whereas in SP they are calculated *inside* the ifFalse: control structure. However, this is a semantics-preserving variation because for SS0012 and MS0014End, the values of price and bookingFee are the same.

- Difference5 (Error): SP carries out a different computation in SS0012 compared to that in MS0014End of MP. This is a semantic variation, which *is* an error in the student program.
- Difference6 (SPV 8): SP uses different numbers of temporary variables in SS0012 and SS0013End. The difference changes the way computations are executed without changing the return values computed.

The challenge in the automatic diagnosis of students' programming errors in this example lies in identifying Difference5 as an error in the student program while accommodating other semantics-preserving changes in the student program.

4. AST program representation

In order to perform program analysis and program transformation automatically, a program must first be represented in AST representation. The generation of the abstract syntax tree for a program is based on the Backus-Naur Forms (BNF) [12] of the programming language in which the program is written. BNF is an application of context-free grammar (CFG) which is defined as G = (V, T, P, S), where V and T are finite sets of variables and terminals respectively. P is a finite set of productions. Every production is in the form of $A \rightarrow \alpha$, where A is a variable and α is a string in the set of (V T)*. S is a special variable, called start symbol. The BNFs of the Smalltalk language are presented below.

Program :: (selector block)
Block :: (*blockArguments body*)
BlockArguments :: (blockArgument ... blockArgument)
Body :: (*temporaries statements*)
Temporaries :: (temporary ... temporary)

Statements :: (statement ... statement)
Statement :: returnStatement | value
ReturnStatement :: (returnMark value)
Value :: number | character | symbol | string | byteArray | array | boolean | undefinedObject | variable | block | cascadeMessage | message | assignment
Assignment :: (variable assignmentMark value)
CascadeMessage :: (value *messages*)
Messages :: (message ... message)
Message :: (value selector *messageArguments* precedence)
MessageArguments :: (value ... value)

Box 4. BNFs for Smalltalk language

In the above representation, non-terminals that should be directly rewritten by a production in which α only contains terminals (i.e. α is a string in the set of (T)*) are underlined. In every production, α is represented as a list. Non-terminals that can be empty lists are shown in italic font. The only start symbol is program.

With the BNFs of the Smalltalk language defined, the parsing tree of a program, which is generated by parsing a program according to the BNFs and the parsing order defined [8], can be produced. An AST representation of a program is a frame-based representation based on the parsing tree together with additional program analysis information such as statement levels and productions used in generating children of the frame. Some variations existing at the source-code level are eliminated in the AST representation. Besides, another good feature of the AST representation for a pure object-oriented programming language such as Smalltalk is that control structures are also represented in regular message forms. For example, ifTrue:Structure :: (anObject ifTrue: aBlock) and to:by:do:Structure :: (anObject to: anObject by: anObject do: aBlock) are productions for two control structures.

For the running example, the simplified representation of SP, called SPTree, is given in Box 5. Nodes in the tree are listed in depth-first traversal order. The BNF applied in the creation of a node is also given beside every node.

l1: (program (#taxiFeeWith:isBookingCase l2))	program :: (selector block)
l2: (block (l3 l4))	block :: (blockArguments body)
l3: (blockArguments (mile isBookingCase))	blockArguments :: (blockArgument blockArgument)
l4: (body (l5 l6))	body :: (temporaries statements)
l5: (temporaries (bookingFee price payment))	temporaries :: (temporary temporary temporary)
l6: (statements (l7 l8 l9))	statements :: (statement statement statement)
l7: (statement l10)	statement :: value
l10: (value l11)	value :: message
l11: (message (l12 #ifTrue:ifFalse: l13 3))	message :: (value selector messageArguments precedence)
l12: (value booking)	value :: variable
l13: (messageArguments (l14 l15))	messageArguments :: (value value)
l14: (value l16)	value :: block
l16: (block (#() l17))	block :: (blockArguments body)
l17: (body (#() l18))	body :: (temporaries statements)
l18: (statements (l19 l20))	statements :: (statement statement)
l19: (statement l21)	statement :: value
l21: (value l22)	value :: assignment
l22: (assignment (price := l23))	assignment :: (variable assignmentMark value)
l23: (value 3.0)	value :: number
l20: (statement l24)	statement :: value
l24: (value l25)	value :: assignment
l25: (assignment (bookingFee := l26))	assignment :: (variable assignmentMark value)
l26: (value 2.0)	value :: number
l15: (value l27)	value :: block
l27: (block (#() l28))	block :: (blockArguments body)
l28: (body (#() l29))	body :: (temporaries statements)
l29: (statements (l30 l31))	statements :: (statement statement)
l30: (statement l32)	statement :: value
l32: (value l33)	value :: assignment
l33: (assignment (price := l34))	assignment :: (variable assignmentMark value)
l34: (value 2.5)	value :: number
l31: (statement l35)	statement :: value
l35: (value l36)	value :: assignment
l36: (assignment (bookingFee := l37))	assignment :: (variable assignmentMark value)
l37: (value 0.0)	value :: number
l8: (statement l38)	statement :: value
l38: (assignment (payment := l39))	assignment :: (variable assignmentMark value)
l39: (value l40)	value :: message
l40: (message (l41 * l42 2))	message :: (value selector messageArguments precedence)
l41: (value l43)	value :: message
l43: (message (price + l44 2))	message :: (value selector messageArguments precedence)
l44: (messageArguments (l45))	messageArguments :: (value)
l45: (value bookingFee)	value :: variable
l42: (messageArguments (l46))	messageArguments :: (value)
l46: (value mile)	value :: variable
l9: (statement l47)	statement :: returnStatement
l47: (returnStatement (^ l48))	returnStatement :: (returnMark value)
l48: (value payment)	value :: variable

Box 5. AST representation for the student program in task taxiFee

Representing programs in AST eliminates SPV2 in programs. In the running example, Difference3, the "." before the "]", is eliminated. With this representation, it is easy to perform various program analyses and manipulations. These are discussed in the following sections.

5. Basic program standardization

Semantics-preserving transformations for basic program standardization do not require DU information. They are performed, if applicable, on both the model program and the student program. In transformation-based diagnosis, there are five basic transformations: *statement separation*, *temporary declaration standardization*, *algebraic expression standardization*, *control structure standardization*, and *Boolean expression standardization*. There are also two advanced transformations: *forward substitution* and *dead code removal*. These two transformations will be discussed in Section 7.

Program transformations are used for different purposes in compiler design, partial evaluation, and the diagnosis of student programs. The purpose of the transformations used in compiler design is to optimize the object code of programs. Transformations are applied to three levels of code: source code, intermediate-level code, and low level code. The purpose of the transformations used in partial evaluation is to translate high-level specifications into programs. Transformations are applied to higher-level specifications as well as to source

code. Program transformations in the diagnosis of student programs should satisfy three criteria: (1) The transformation should be applicable to programs at source code level. (2) The transformation should be convergent in order to avoid cyclic triggering of transformations. (3) The transformation should be an intra-procedural transformation because the scope of the diagnosis handled in this study is limited to intra-procedural diagnosis.

The transformations used in transformation-based diagnosis were identified from a study of all program transformations used in compiler design and partial evaluation. For example, folding and unfolding are common program transformations in partial evaluation. They are called common-expression elimination and copy propagation respectively in compiler design. However, for the diagnosis of student programs, only one of them can be used for program standardization. If both of them are used, cyclic triggering of transformations will arise. In transformation-based diagnosis, copy propagation (i.e., unfolding) is used for program standardization, and we call it forward substitution.

Statement separation

When cascaded messages are used in the program, they may make comparison difficult. Cascaded messages can be standardized to a message sequence.

Example

```
x := list new: 2.                    x := list new: 2.
x at: 1 put 1;                       x at: 1 put 1.
  at: 2 put: 4.                      x at: 2 put: 4.
```

Temporary declaration standardization

Temporary variables can be defined both in the method temporary variable declaration and in the block temporary variable declaration. Standardization for temporary variables is to let all temporary variables be defined only in the method temporary variable declaration.

Example

```
| listSize |                                  | listSize item |
listSize := self size.                     listSize := self size.
1 to: (listSize) do: [ :i |                1 to: (listSize) do: [ :i |
    | item |                                   item := (self at: i)*(self
at: i).
        item := (self at: i)*(self at: i).       self at: i put:
item ].
    self at: i put: item ].                  ^self
^self
```

Algebraic expression standardization

An algebraic expression is defined as t1 binarySelector t2, where t1, t2 are objects or expressions, and binarySelector is '+', '-', or '*'. Algebraic expression standardization makes use of the properties of the binary selectors—namely, associativity, commutativity, and distributivity—and the weights of t1 and t2 denoted as weight(t1) and weight(t2) respectively. They are constants calculated according to the following rules.

(1) weight (t1 binarySelector t2) := weight(t1) + weight(t2), where t1 and t2 are any algebraic objects or algebraic expressions.

(2) weight (c) < weight (nc), where c is a constant and nc is a non-constant.

The procedure for standardizing an algebraic expression is to apply the following 11 transformation rules on the algebraic expression according to *the order* of the rules listed repeatedly until no further transformation rules can be applied. In the rules below, t1, t2, and t3 are algebraic objects or algebraic expressions.

Rule1(division):	t1 / t2	→	t1 * (1 / t2)
Rule2(commutativity):	t2 + t1	→	t1 + t2, where weight(t1)<weight(t2)
Rule3(commutativity):	t2 * t1	→	t1 * t2, where weight(t1)<weight(t2)
Rule4(commutativity):	t2 – t1	→	-t1 + t2, where weight(t1)<weight(t2)
Rule5(associativity):	(t1 + t2) + t3	→	t1 + (t2 + t3)
Rule6(associativity):	(t1 * t2) * t3	→	t1 * (t2 * t3)
Rule7(associativity):	t2 + (t1 + t3)	→	t1 + (t2 + t3), where weight(t1)<weight(t2)
Rule8(associativity):	t2 * (t1 * t3)	→	t1 * (t2 * t3), where weight(t1)<weight(t2)
Rule9(distributivity):	(t1 + t2) * t3	→	t1 * t3 + t2 * t3
Rule10(distributivity):	t1 * (t2 + t3)	→	t1 * t2 + t1 * t3
Rule11(distributivity):	(t1 - t2) * t3	→	t1* t3 - t2 * t3
Rule12(distributivity):	t1 * (t2 - t3)	→	t1 * t2 - t1 * t3

Example x * (y + 2) → x * (2 + y) → x * 2 + x * y → 2 * x + x * y,

where rule1, rule9, and rule 2 are applied in sequence.

Control structure standardization

Control structure standardization standardizes all control structures into one of three structures—ifTrue:ifFalse:, whileTrue:, and to:by:do:. The 11 transformation rules used are given below, where receiver is an object, and b1, b2 are blocks.

Rule1:	receiver ifTrue: b1	→	receiver ifTrue: b1 ifFalse: [ ]
Rule2:	receiver ifFalse: b1	→	receiver ifTrue: [ ] ifFalse: b1
Rule3:	receiver ifFalse: b1 ifTrue: b2	→	receiver ifTrue: b2 ifFalse: b1
Rule4:	receiver and: b1	→	receiver ifTrue: b1 ifFalse: [false]
Rule5:	receiver or: b1	→	receiver ifTrue: [true] ifFalse: b1
Rule6:	receiver whileFalse: b1	→	receiver not whileTrue: b1
Rule7:	receiver whileFalse	→	receiver not whileTrue: [ ]
Rule8:	receiver whileTrue	→	receiver whileTrue: [ ]
Rule9:	receiver repeat	→	[true] whileTrue: receiver
Rule10:	receiver to: limit do: b1	→	receiver to: limit by: 1 do: b1
Rule11:	receiver timesRepeat: b1	→	1 to: receiver by: 1 do: b1

Boolean expression standardization

A Boolean expression is defined as t1 binarySelector t2, where t1, t2 are Boolean objects (e.g., Boolean values, variables, or conditional expressions) or Boolean expressions, and binarySelector is a Boolean operator such as '|', '&', or 'not'. Boolean expression standardization makes use of the weights of t1 and t2 denoted as weight(t1) and weight(t2) respectively. They are constants calculated according to the following rules.

(1) weight (t1 binarySelector t2) := weight(t1) + weight(t2), where t1 and t2 are any Boolean objects or Boolean expressions.

(2) weight (c) < weight (nc), where c is a constant and nc is a non-constant.

(3) weight (t1) < weight (not t2), where t1 and t2 are any Boolean objects or Boolean expressions.

The procedure for standardizing a Boolan expression is to apply the following 18 transformation rules on the Boolean expression according to *the order* of the rules listed repeatedly until no further transformation rules can be applied. In the rules below, t1, t2, and t3 are Boolean objects or Boolean expressions.

Rule1(identity):	t1 | t1	→	t1
Rule2(identity):	t1 & t1	→	t1
Rule3(identity):	not (not t1)	→	t1
Rule4(identity):	t1 & not t1	→	false
Rule5(identity):	t1 | not t1	→	true
Rule6(identity):	t1 | true	→	true
Rule7(identity):	t1 & false	→	false
Rule8(commutativity):	t2 | t1	→	t1 | t2, where weight(t1)<weight(t2)
Rule9(commutativity):	t2 & t1	→	t1 & t2, where weight(t1)<weight(t2)
Rule10(associativity):	(t1 | t2) | t3	→	t1 | (t2 | t3)
Rule11(associativity):	(t1 & t2) & t3	→	t1 & (t2 & t3)
Rule12(associativity):	t2 | (t1 | t3)	→	t1 | (t2 | t3), where weight(t1)<weight(t2)
Rule13(associativity):	t2 & (t1 & t3)	→	t1 & (t2 & t3), where weight(t1)<weight(t2)
Rule14(identity): t1 | t2	→	(t1 & t2) | (t1 & not t2) | (not t1 & t2), where t1,	

t2 are Boolean objects

Rule15(identity): not (t1 | t2) → not t1 & not t2

Rule16(identity): not (t1 & t2) → (not t1 & not t2) | (not t1 & t2) | (t1 & not t2)

Rule17(distributivity): (t1 | t2) & t3 → (t1 & t3) | (t2 & t3)

Rule18(distributivity): t1 & (t2 | t3) → (t1 & t2) | (t1 & t3)

Example ((z>0) | (y>0)) & (x>0)

→ (t3 | t2) & t1

→ t1 & (t3 | t2) apply Rule9

→ t1 & (t2 | t3) apply Rule8

→ t1 & ((t2 & t3) | (t2 & not t3) | (not t2 & t3)) apply Rule14

→ t1 & (t2 & t3) | t1 & (t2 & not t3) | t1 & (not t2 & t3) apply Rule18

Using the basic standardization transformations, SPV3, SPV4, SPV5, SPV6, and SPV7 are eliminated in programs. In the running example, basic standardization transformations including algebraic expression standardization and control structure standardization are applied on MPTree. The algebraic expression standardization is applied on SPTree. The standardized MP and SP are shown in Box 6 and Box 7.

```
MEntry      taxiFeeWith: mile isBookingCase: bookingCase
            | bookingFee price |
MS0011      bookingFee := 0.0.
MS0012      price :=2.5.
MS0013      (bookingCase)
MS1321          ifTrue: [bookingFee := 2.0.
MS1322                  price := 3.0 ]
            ifFalse: [nil].
MS0014End    ^(price *mile + bookingFee)
```

Box 6. The model program after applying the basic standardization transformations

```
SEntry      taxiFeeWith: mile isBookingCase: booking
            | bookingFee price payment |
SS0011      (booking)
SS1121          ifTrue: [ price := 3.0.
SS1122                  bookingFee := 2.0]
SS1123          ifFalse: [ price :=2.5.
SS1124                    bookingFee := 0.0].
SS0012          payment := price *mile+ (bookingFee * mile).
SS0013End       ^payment
```

Box 7. The student program after applying the basic standardization transformations

6. AOFG program representation

In order to compare SP and MP at the semantic level, a program representation that presents the semantics of programs is required. Program Representation Graphs (PRGs) are used by Yang [27] in a program-integration algorithm to facilitate the semantic level program comparison. It combines features of Program Dependence Graphs (PDGs) [6] and Static-Single-Assignment (SSA) forms. A PDG consists of a Control Dependence Subgraph (CDS) and a Data Dependence Subgraph (DDS). It provides the static compile-time semantics of the program. In the PRG, a CDS is augmented as follows. A vertex labeled "φinitial: x := initial(x)" is added to the beginning of the CDS for each variable x that may be used before being defined. As in SSA forms, the CDS is further augmented with special "φ vertices" so that each use of a variable in a statement is reached by exactly one definition. However, the PDG representation needs to be extended for object-oriented programs.

An object-oriented extension of the PDG representation, called the Object-oriented Program Dependence Graph (OPDG), is described by McGregor [19]. The OPDG representation extends the PDG for object-oriented programs by representing the class inheritance structure, using object-based dependence graph, and representing the dynamic, runtime aspects of an object-oriented program as an Object Dependence Subgraph. However, the

extension is only applicable for programs in C++. For programs written in a pure object-oriented programming language such as Smalltalk, many additional issues must be considered. One problem that arises is, in Smalltalk programs, several statements can be embedded in a conditional block in a loop statement; this is not allowed in C++. We illustrate this below, where S1, S2, S3, and S4 are statements that may even contain conditional or loop structures, and V1 is a variable. One Smalltalk statement corresponds to several C++ statements.

In Smalltalk: [S1. S2. V1] whileTrue: [S3. S4].

In C++: S1; S2; while (V1) do: [S3; S4; S1; S2];

To overcome this problem in transformation-based diagnosis, an improved flow graph representation, called Object-oriented Flow Graph (OFG), is used in producing the OPDG. As in the case with a PRG, an OFG is also augmented with various φ vertices and becomes an Augmented OFG (AOFG). An AOFG is used to generate the Object-oriented Augmented Control Dependence Subgraph (AOCDS) and the Object-oriented Augmented Data Dependence Subgraph (AODDS) in a AOPDG, which is used for the semantic level program comparison.

Object-oriented Flow Graph (OFG) representation

In the OFG, a new edge coming out from the predicate node points to the statements in the conditional block. The OFG for the above statement in Smalltalk is shown in Fig. 2.

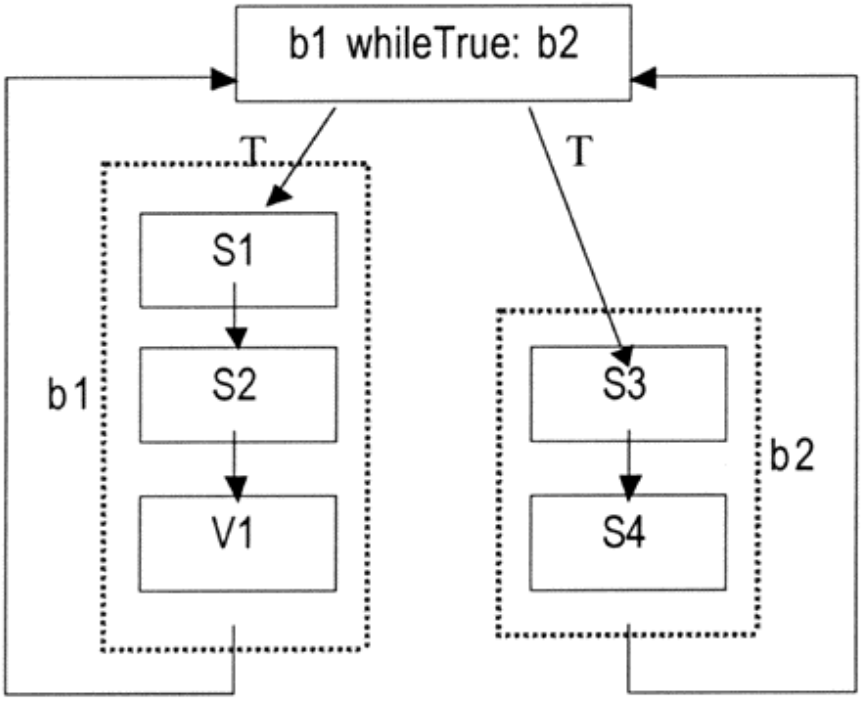


Fig. 2. Object-oriented Flow Graph for a statement in Smalltalk

Augmented Object-oriented Flow Graph (AOFG) representation

To construct an AOFG from an OFG, a vertex labeled "φ initial: x := initial(x)" is added at the beginning of the OFG for each variable x that may be used before it is defined. A φ vertex labeled "φ enter x := x" is added inside each loop statement immediately before the loop predicate for each variable x that is defined within the loop, and it is live immediately before the loop predicate (i.e., x may be used either inside the loop, after the loop, or by the loop predicate before being redefined). A vertex labeled "φ exit x := x" is added immediately after the loop for each variable that is defined within the loop and is live after the loop.

In the running example, the AOFG of the student program is shown in Fig. 3.

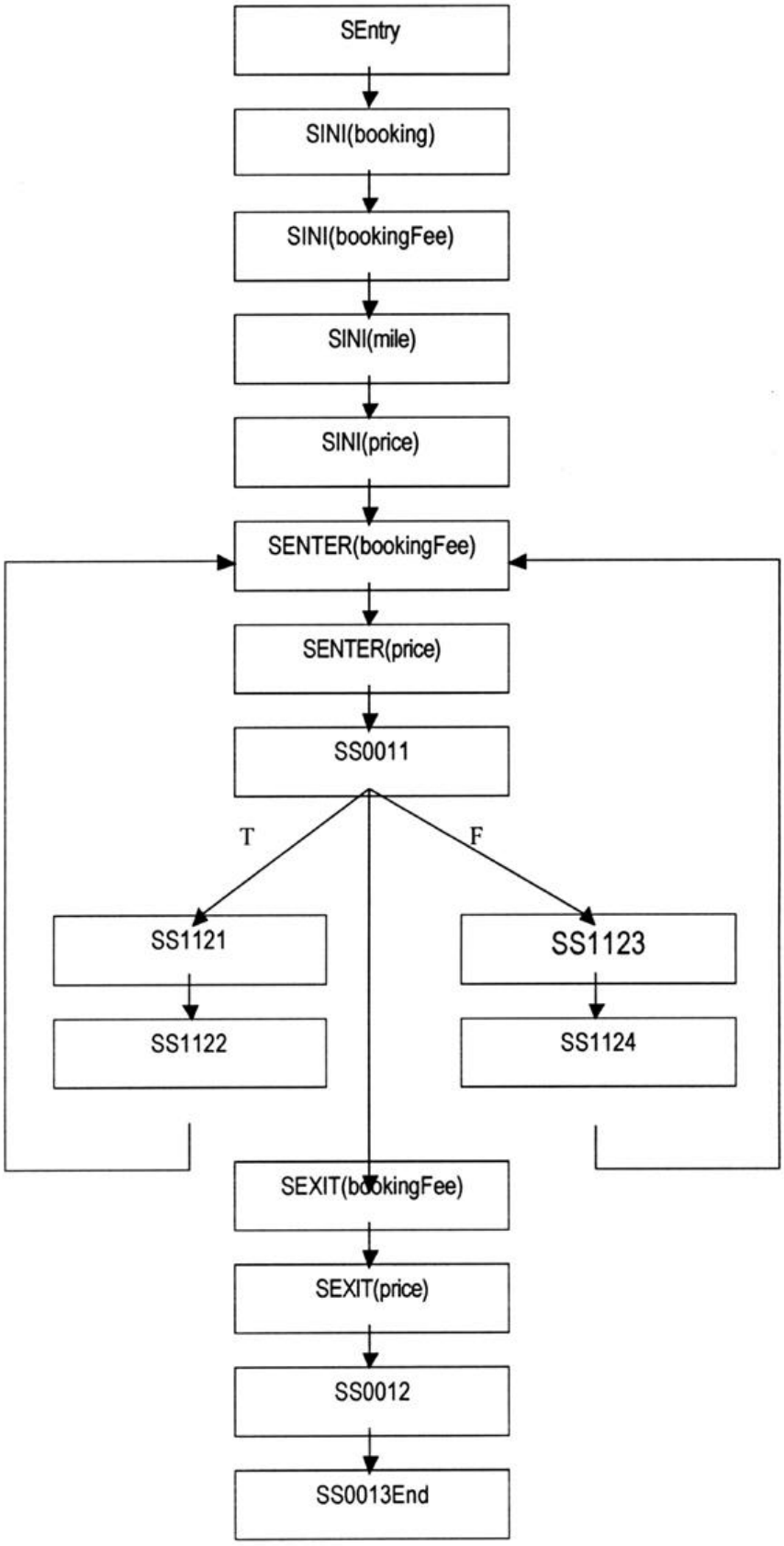


Fig. 3. The AOFG for the student program in task taxiFee

In Fig. 3, SINI(bookingFee) is an augmented vertex labeled φinitial: booingFee := initial(bookingFee). Similarly, SENTER(bookingFee) is an augmented vertex labeled φenter: bookingFee := bookingFee, and SEXIT(bookingFee) is an augmented vertex labeled φexit: bookingFee := bookingFee.

Definition-use information calculation

Definition-use information of a program is required for generating the dependence graph of the program. It is also required for the advanced standardization transformations, which are discussed later. The calculation of the DU information for a program is based on its flow graph [20].

For a vertex named x, RCHin(x) represents the set of definitions valid on entry to x, and RCHout(x) represents the set of definitions reaching the end of x. Initially, RCHin(i) = φ, for all i. RCHin(x) and RCHout(x) are calculated according to the following equations.

$$RCHout(i) = GEN(i) \quad (RCHin(i) \cap PRSV(i)) \text{ for all } i \qquad (1)$$

$$RCHin(I) = \boxed{\times} RCHout(j) \text{ for all } i \qquad (2)$$

In the above equation, GEN(i) is the set of definitions generated by vertex i, and PRSV(i) is the set of definitions preserved by vertex i. They are obtained from the program analysis performed on AST representations of the programs. To calculate DU information for a program, RCHin(x) and RCHout(x) are calculated for every vertex in the flow graph repeatedly until there are no further changes in the values of RCHin(x) and RCHout(x).

7. Advanced program standardization

In order to eliminate variations such as SPV8, program transformations—forward substitution and dead code removal—for advanced program standardization are used.

Forward substitution

Let A be an assignment such as a := f(b, c), where a is a temporary variable *assigned to* a value calculated by the expression f(b, c), in which b and c are *referred to*. If there is a statement S that satisfies the following conditions, the forward substitution transformation can be applied by simply replacing a in S with f(b, c).

(1) a is not defined in S.

(2) The definition of a by A is alive in RCHin(S).

(3) The definitions of b and c alive in RCHin(A) are still alive in RCHin(S)

If a is not *referred to* after the program is applied with the forward substitution transformation, then assignment A, as a SPV9, will be removed as dead code by the transformation of dead code removal.

Dead code removal

A dead statement is defined as a statement that satisfies the following conditions:

(1) It does not have any outgoing control or data dependence edges

(2) It is not a return statement

Dead code removal removes dead statements identified in a program from the AST representation of the program. If a temporary variable is not used in the program, the declaration of the variable is also removed.

For the purpose of standardization, the effects of applying the forward substitution and dead code removal are the same as many optimization transformations in compiler design, such as constant-expression evaluation, copy propagation, sparse conditional constant propagation, and redundancy elimination and reassociation [17].

In the running example, no advanced standardization transformation is applicable to MP. For SP, SS0013End is changed by forward substitution, and SS0012 is removed by dead code removal. Difference6 in SP is eliminated. SP standardized by advanced standardization transformations is shown in Box 8.

```
SEntry      taxiFeeWith: mile isBookingCase: booking
            | bookingFee price payment |
SS0011      (booking)
SS1121          ifTrue: [ price := 3.0.
SS1122                  bookingFee := 2.0]
SS1123          ifFalse: [ price :=2.5.
SS1124                   bookingFee := 0.0].
SS0012End   ^ price *mile+ (bookingFee * mile)
```

Box 8. The student program after applying the advanced standardization transformations

8. AOPDG program representation

In order to compare the programs at the semantic level, programs are represented in AOPDGs. An AOPDG consists of an AOCDS and an AODDS. Similar to the construction of a CDS and a DDS from a flow graph [20], AOCDS and AODDS are constructed from an AOFG.

In AOCDS representation, there are six types of vertices: entry vertex, end vertex, statement vertex, initialization φ vertex, enter φ vertex, and exit φ vertex. The source of a control edge in AOCDS is always either an entry vertex or a predicate vertex (i.e. a conditional statement vertex).

There are five types of control dependence edges:

(1) controltrue: an edge from a predicate vertex to a vertex in the true branch of the predicate vertex, or a normal control edge.

(2) controlfalse: an edge from a predicate vertex to a vertex in the false branch of the predicate vertex.

(3) controlloop: an edge from a predicate vertex to itself.

(4) entertrue: an edge from a predicate vertex to an enter φ vertex in the true branch of the predicate vertex, or a normal control edge to an enter φ vertex.

(5) enterfalse: an edge from a predicate vertex to an enter φ vertex in the false branch of the predicate vertex.

The types of data dependence edges are as follows.

- flow1: an edge from a vertex containing a definition of a variable x to another vertex S that uses x, where x is the first operand in S.
- Similarly for flow2, flow3, ..., flown.
- flowtrue: an edge from a definition in the true branch of a predicate to the enter φ vertex corresponding to the predicate vertex.
- flowfalse: an edge to the enter φ vertex of a predicate vertex. The edge may come from a definition in the false branch of the predicate or come from a non-φ vertex definition out of the branches of the predicate.
- flowenter: an edge from a φ vertex to an enter φ vertex.
- flowexit: an edge from an enter φ vertex to an exit φ vertex.
- flowwhile: an edge from a predicate vertex to its corresponding exit φ vertex.

By representing the programs in AOPDGs, SPV10 in programs is eliminated. The advanced standardization of dead code removal is applied after the AOPDG of a program is generated. A dead statement is defined as follows: (1) the statement that does not have any outgoing control or data dependence edges; and (2) the statement is not a return statement. Whenever dead code is identified in a program, the operation of dead code removal is carried out on the AST representation of the program.

In the running example, the AOPDG representations of SP and MP are called SPGraph and MPGraph, and they are depicted in Fig. 4 and Fig. 5 respectively. For SPGraph shown in

Fig. 4, the vertices are the nodes in the student AOFG shown in Fig. 3. Control dependence edges are depicted in solid lines, and the control dependence types are appropriately labeled. Data dependence edges are depicted in dotted lines, and the data dependence types are appropriately labeled. Difference2 in SP is eliminated by representing SP and MP in AOPDGs.

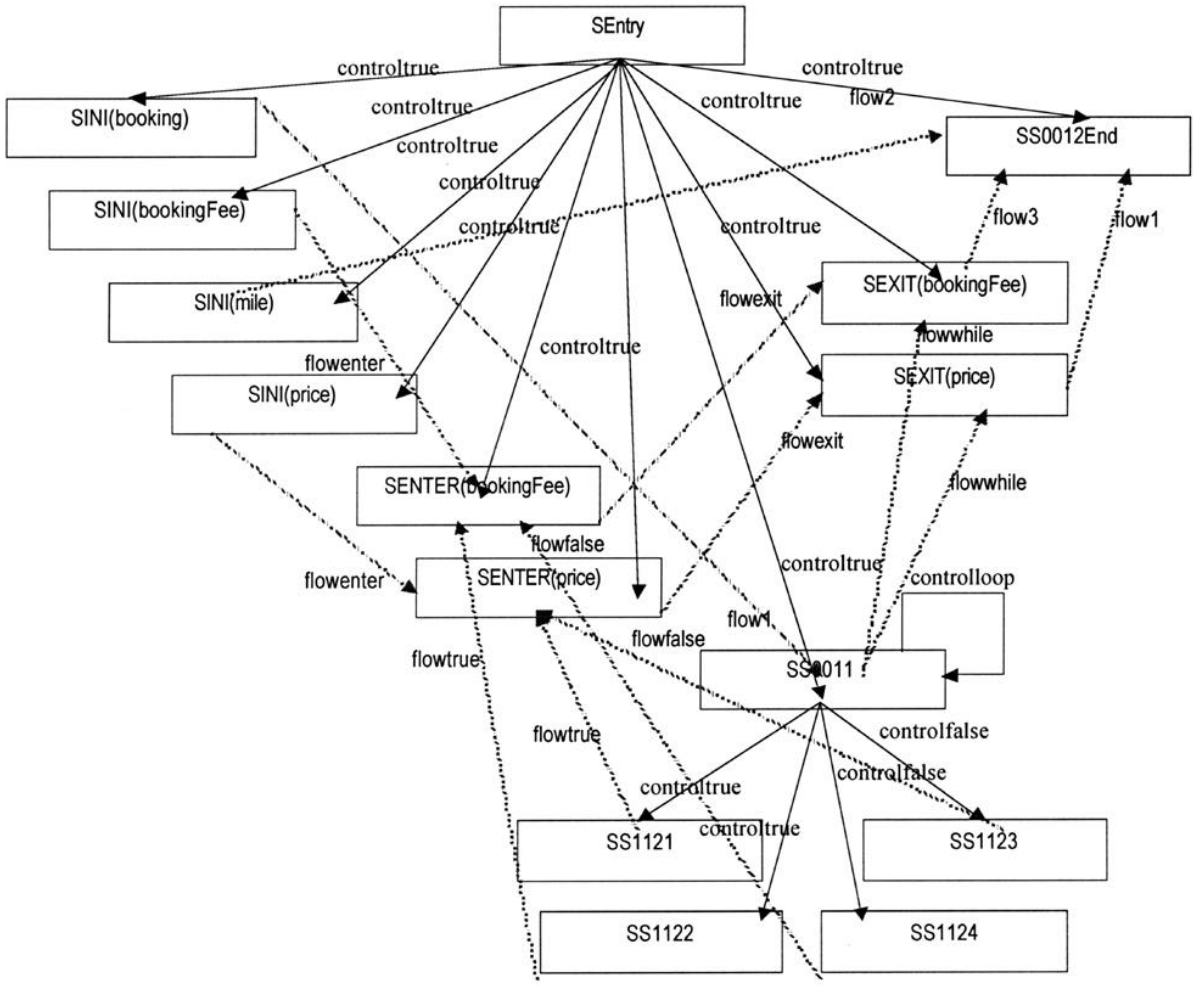


Fig. 4. SPGraph in task taxiFee

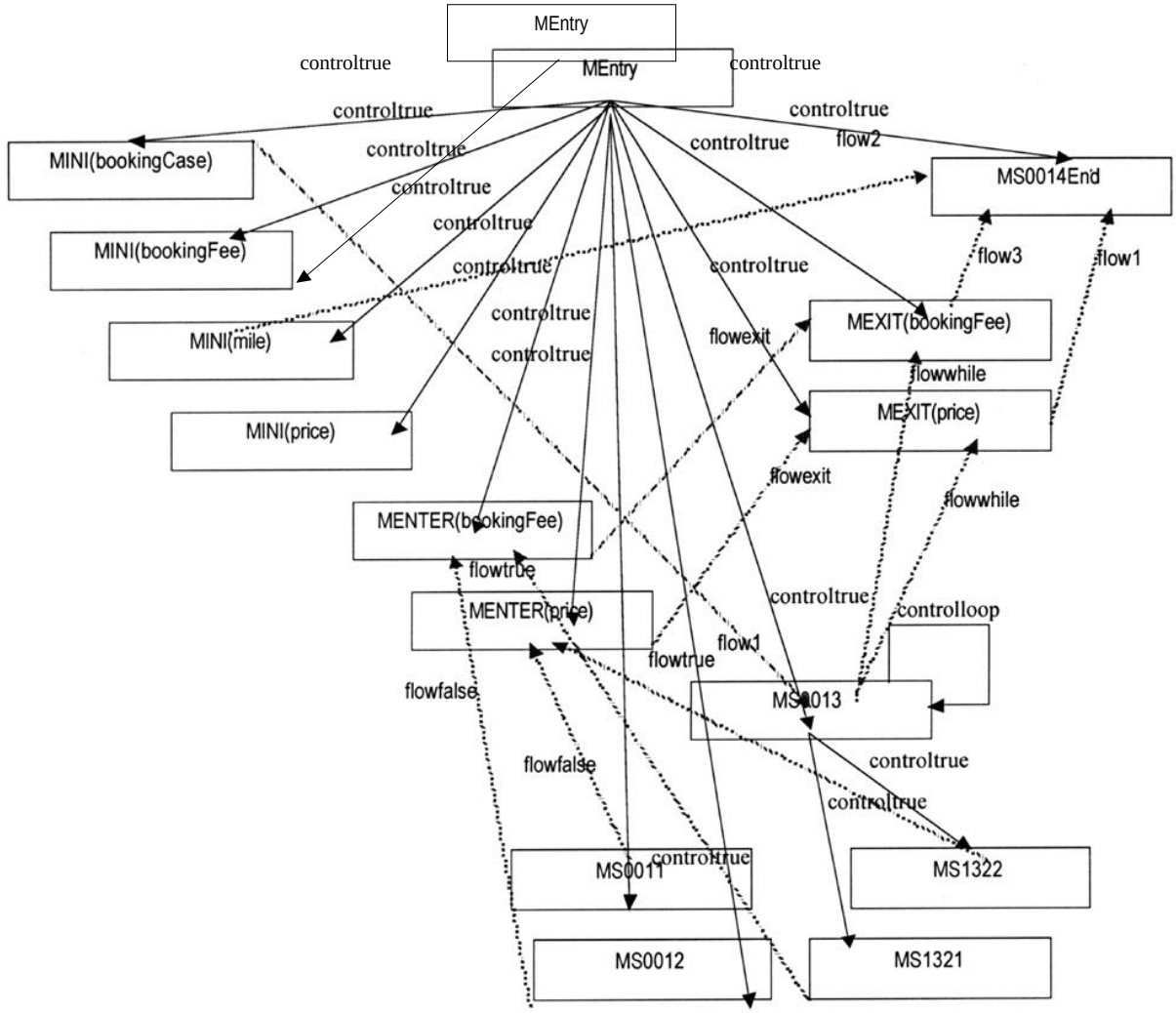


Fig. 5. MPGraph in task taxiFee

9. Program comparison

After SP and MP are standardized by basic and advanced standardization transformations and presented in AOPDGs, they are compared at the semantic level with a partitioning algorithm [27]. First, an initial partition is created. Vertices in SPGraph and MPGraph are classified into different sets in the initial partition according to the operators and the

operands in the vertices. Next, the initial partition is refined by a basic partitioning algorithm by considering data dependence edges. The idea in the basic partitioning algorithm is that a set in a partition containing several vertices whose predecessors belong to different sets in the partition must be split into smaller sets according to the partitions of their predecessors. There exists a coarsest refinement of the initial partition that is stable [27].

It can be proved that if a vertex in SPGraph and a vertex in MPGraph are in the same set in the refined partition, the two statements in the vertices are surely semantically equivalent. With the control dependence information represented in the labels of the data dependence edges, one pass of the refinement upon the data dependence edges makes the comparison accommodate semantics-preserving differences caused by using different program logical structures. SPV12 is recognized in the program comparison. Based on the refined partition, three sets are calculated as the results of the comparison. They are the *equivalent map,* the *textual difference map,* and the *unmatched map,* which were explained in section 2.4.

In the running example, the initial partition and the refined partition are shown in Box 9 and Box 10 with the names of the partition sets marked on the left. The equivalent map, the textual difference map, and the unmatched map are also shown in Box 11, Box 12, and Box 13 respectively.

{taxiFeeWith:isBookingCase:	→	(MEntry SEntry)
INI	→	(all initialization φ vertices SP and MP)
ENTER	→	(MENTER(bookingFee) MENTER(price) SENTER(bookingFee) SENTER(price))
EXIT	→	(MEXIT(bookingFee) MEXIT(price) SEXIT(bookingFee) SEXIT(price))
0.0	→	(MS0011 SS1124)
2.5	→	(MS0012 SS1123)
ifTrue:ifFalse:	→	(MS0013 SS0011)
3.0	→	(MS1322 SS1121)
2.0	→	(MS1321 SS1122)
+	→	(MS0014End SS0012End)}

Box 9. The initial partition in task taxiFee

{taxiFeeWith:isBookingCase:	→	(MEntry SEntry)
INI	→	(all initialization φ vertices SP and MP)
ENTER	→	(MENTER(bookingFee) MENTER(price))
EXIT	→	(MEXIT(bookingFee) MEXIT(price))
0.0	→	(MS0011 SS1124)
2.5	→	(MS0012 SS1123)
ifTrue:ifFalse:	→	(MS0013 SS0011)
3.0	→	(MS1322 SS1121)
2.0	→	(MS1321 SS1122)
+	→	(MS0014End)
NEW1	→	(SENTER(bookingFee) SENTER(price))
NEW2	→	(SEXIT(bookingFee) SEXIT(price))
NEW3	→	(SS0012End)}

Box 10. The refined partition in task taxiFee

{taxiFeeWith:isBookingCase:	→	(MEntry SEntry)
0.0	→	(MS0011 SS1124)
2.5	→	(MS0012 SS1123)
ifTrue:ifFalse:	→	(MS0013 SS0011)
3.0	→	(MS1322 SS1121)
2.0	→	(MS1321 SS1122)}

Box 11. The equivalent map in task taxiFee

{taxiFeeWith:isBookingCase:	→	(MEntry SEntry)
ifTrue:ifFalse:	→	(MS0013 SS0011)}

Box 12. The textual difference map in task taxiFee

{+	→	(MS0014End)
NEW3	→	(SS0012End)}

Box 13. The unmatched map in task taxiFee

Difference1 is correctly identified as textual difference in this step because taxiFeeWith:isBookingCase: →(MEntry SEntry) and ifTrue:ifFalse:→(MS0013 SS0011) are in the textual difference map. Difference4 is accommodated in the comparison because

0.0→(MS0011 SS1124) and 2.5→(MS0012 SS1123) are in the equivalent map. Difference5 is identified as a semantic difference between SP and MP because +→(MS0014End) and NEW3→(SS0012End) are in the unmatched map.

10. Error detection

The results of the comparison report those statements in SP that contain semantic differences. However, a programming tutoring system should be able to pinpoint the errors in the incorrect statements and to provide corrections of the errors. The purpose of the error detection step is to identify semantic errors in the unmatched student statements, explain the errors, and provide the corrections to the errors.

10.1 Handling textual differences

In order to identify semantic errors in a SP, it is necessary to first identify and eliminate textual differences between SP and MP. Specifically, textual differences between a SP statement and a MP statement in the textual difference map are pinpointed by comparing them at the syntax level. The model program is then changed according to the pinpointed textual differences, and the programs are compared again. In the running example, Difference1 in SP (i.e., SPV11) is eliminated.

10.2 Pinpointing semantic differences

After the elimination of the textual differences, semantic differences between the unmatched statements are pinpointed. For every unmatched SP statement, the most similar MP statement in the unmatched map is found. Pairs of similar statements are put into a list called *Similar Statement Map* (SSM). The definition of similarity is based on the following factors: (1) operators, (2) operands, (3) incoming control dependence edges, (4) outgoing control dependence edges, (5) incoming data dependence edges, and (6) outgoing data dependence edges.

Semantic differences between a similar unmatched statement pair are pinpointed by comparing them at the syntax level. The differences are presented using statement component pairs and put into a list called *Semantic Difference Map* (SDM). Any unmatched MP statements are regarded as extra statements in MP and put into a list called *Extra Statement Set* (ESS).

10.3 Learning semantics-preserving variations

With careful study, we find that not all the semantic differences identified so far are genuine errors in SP. Because the differences identified above are *operational semantic differences* rather than *computational semantic differences*, some differences in the semantic difference map may still be semantics-preserving variations (e.g., SPV13 due to using different statements), which remain after program standardization and program matching. Therefore,

we introduce a variation-learning process as a last resort for the diagnosis system to deal with residual variations of this nature. This idea of the variation-learning process is that the system is able to learn sporadic or unsystematic semantics-preserving variations as a form of heuristic knowledge, which will then be applied in appropriate situations.

The detailed process of the variation learning is as follows. After obtaining the semantic difference map for a statement pair (*S, S'*), the system classifies each pair (*c, c'*) in the semantic difference map into one of the three categories: *equivalent component pair*, *context-equivalent component pair*, and *nonequivalent pair*. It is important to note that the self-learning process occurs only during the training stage of the system. It does not occur in the diagnosis stage of the system.

Equivalent component pair

Components in an equivalent component pair are equivalent in any situation. Equivalent component pairs are recorded in a set called EqualComponents. An example is the component pair ({2.5 * mile} {0.0 + (2.5 * mile)}), where (2.5 * mile) is used in a SP statement and (0.0 + (2.5 * mile)) is used in a MP statement. If the system finds this pair in EqualComponents, SP is modified to match that used in MP. Otherwise, the system asks the teacher which category the pair belongs to. If the teacher identifies the pair as an equivalent component pair, the system changes MP to follow that used in SP and records the pair in the set EqualComponents. The component pair is then removed from the semantic difference map.

Context-based equivalent component pair

Components in a context-based equivalent component pair are equivalent only in a specific context. Context-based equivalent component pairs are recorded in a set called ContextBasedEqualComponents. An example is the component pair ({0.5 + price} {3.0}), where (0.5 + price) is used in SP and 3.0 is used in MP. The context of SP indicates that at the point of evaluating the component (0.5 + price), the value of price is 2.5. The system records this pair together with its context, (S108 ({0.5 + price} 3.0)) where S108 is the name of SP, in the set ContextBasedEqualComponents. The component pair is also removed from the semantic difference map.

Nonequivalent pair

Nonequivalent pairs are recorded in a set called NonEqualComponents. An example is the component pair ({mile * price} {bookingFee}). The pairs in this category are genuine semantic errors in SP. The left-hand side of a pair pinpoints an error in SP, and the right-hand side of the pair shows the correction necessary for the error. The system leaves the nonequivalent pair in the semantic difference map.

10.4 Detecting errors and producing diagnosis reports

With the inclusion of this variation-learning process, SPV13 is eliminated. Thus, all semantic differences between the student and the model program that remain after the previous processes have been executed are genuine errors in the student program. The purpose of error detection is to produce a diagnosis report. A diagnosis report indicates whether SP is

correct. If SP is incorrect, the report identifies the errors in SP and provides explanations of the errors. The report also indicates whether every statement is correct. If a statement is incorrect, the report provides explanations of the errors.

The reasoning for producing a diagnosis report is based on the following information: (1) The Unmatched Map (UM). (2) The Semantic Difference Map (SDM) for every unmatched statement. (3) The Extra Statement Set (ESS) in MP. (4) The Similar Statement Map (SSM). The details of the reasoning used by the system to detect types of errors and produce diagnosis reports are described in Table 1 and Table 2.

TABLE 1

Reasoning for the diagnosis of the student program

Reasoning	*Correctness of SP*	*Error indication*
UM = φ	Correct	NA
(UM ≠ φ) & (i{SDM(i) = φ}) & (ESS ≠ φ)	Incorrect1	Extra statements in MP
(UM ≠ φ) & (i{SDM(i) = φ}) & (ESS = φ)	Incorrect2	Incorrect control structure in SP
(UM ≠ φ) & (i{SDM(i) ≠ φ}) & (ESS = φ)	Incorrect3	Incorrect expression in SP

TABLE 2

Reasoning for the diagnosis of a statement S(i) in the student program

Correctness of SP	*Reasoning*	*Correctness of S(I)*	*Error indication*	*Correction*
Correct	NA	Correct	NA	NA
Incorrect1	S(i) SSM	Correct	NA	NA
	S(i) SSM	Incorrect	Extra statements in MP	ESS
Incorrect2	S(i) SSM	Correct	NA	NA
	S(i) SSM	Incorrect	Incorrect control structure elsewhere	NA
Incorrect3	S(i) SSM	Correct	NA	NA
	S(i) SSM SDM(i) = φ	Incorrect	Incorrect expression elsewhere	NA
	S(i) SSM SDM(i) ≠ φ	Incorrect	Incorrect expression in S(i)	SDM

In the running example, the pinpointed textual difference is ({booking} {bookingCase}) and the model program is changed accordingly. The SSM is {(MS0014End SS0012End)}. The SDM for the only pair in SSM is {({price * mile} {bookingFee}) ({mile} {price}) ({bookingFee} {mile})}. No SPV is learned and applied in this example. The diagnosis report for the running example is given in Box 14.

```
====================================Diagnosis
Report====================================
Student Name: S1       algorithm1       INCORRECT (expression errors)
SENTRY taxiFeeWith:isBookingCase:#({mile} {booking})
MENTRY taxiFeeWith:isBookingCase:#({mile} {booking})----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS0011 {booking ifTrue: aBlock ifFalse: aBlock}
MS0013 {booking ifTrue: aBlock ifFalse: aBlock}----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS1121 {price := 3.0}
MS1322 {price := 3.0}----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS1122 {bookingFee := 2.0}
MS1321 {bookingFee := 2.0}          ----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS1123 {price := 2.5}
MS0012 {price := 2.5}----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS1124 {bookingFee := 0.0}
MS0011 {bookingFee := 0.0}          ----------CORRECT
-------------------------------------------------------------------------------------------------------------------------------------------
-------
SS0012END {^price * mile + (mile * bookingFee)}
MS0014END {^bookingFee + (price * mile)}----------INCORRECT (expression errors in this statement).
{price * mile} → {bookingFee}
{mile}→{price}
{bookingFee}→{mile}
```

Box 14. The diagnosis report of the task taxiFee

In Box 14, every SP statement is listed together with a corresponding MP statement. Statement references are also shown to the left of the statements. The student statement SS0012END {^bookingFee + (price * mile)} is identified to be incorrect. Difference5 is identified

as the semantic error in the incorrect student statement. The report indicates that the type of the error is "expression error in this statement." Corrections for the error are provided as {price * mile} → {bookingFee}, {mile}→{price}, and {bookingFee}→{mile}. Based on such information from the diagnosis report, a programming tutoring system can provide situated help to the student when he or she submits a program to the tutoring system.

11. Summary of variations and handling strategies

All 13 semantics-preserving variations except SPV1 are handled by the various strategies discussed above. Transformation-based diagnosis handles SPV1 (i.e., using different algorithms in the student programs) by using different model programs that correspond to the different algorithms used in the student programs. In diagnosis, a proper model program is selected for a student program based on a best-matching criterion. This issue is elaborated on in Section 12. A summary of the handling strategies for all the semantics-preserving variations listed in Section 2 is given in Table 3.

TABLE 3

Handling strategies for all semantics-preserving variations

SPV	*Handling strategy*
SPV1	Use different model programs
SPV2	Represent programs in ASTs
SPV3	Apply basic standardization transformation—statement separation
SPV4	Apply basic standardization transformation—temporary variable declaration
SPV5	Apply basic standardization transformation—algebraic expression standardization
SPV6	Apply basic standardization transformation—control structure standardization
SPV7	Apply basic standardization transformation—Boolean expression standardization
SPV8	Apply advanced standardization transformation—forward substitution
SPV9	Apply advanced standardization transformation—dead code removal
SPV10	Represent programs in AOPDGs
SPV11	Identify textual differences in the comparison and change model programs accordingly
SPV12	Be recognized in the comparison
SPV13	Learn and apply statement component-pair sets

Table 3 shows that all semantics-preserving variations can be properly handled properly using transformation-based diagnosis. Any semantic differences that remain in the student program are identified as programming errors in the diagnosis report.

Transformation-based diagnosis is a conservative and safe approach to the diagnosis of student programs. It is possible that the system falsely reports an error. This situation can arise when a semantics-preserving variation is not identified as such because it is not recognized as an equivalent component pair or a context-based equivalent component pair (described in Section 10.3). However, it is impossible for the system to miss an error if one actually exists.

12. Empirical evaluation

Transformation-based diagnosis is implemented using Smalltalk/VisualWorks 2.5 in *SIPLeS-II*, a system for the diagnosing student programs written in Smalltalk. The approach of transformation diagnosis is general. It can be implemented in Pascal, C/C++, and Java to diagnose programs written in these programming languages.

12.1 A use scenario

A scenario of using *SIPLeS-II* to diagnose student programs written for a programming task, TaskA, is described below.

(1) *Model program accumulation and variation learning*: A model program given by the instructor is used to diagnose a set of student programs. Based on the diagnosis reports, student programs using different algorithms, compared to that used in the model program, are identified. More model programs corresponding to the different algorithms are given by the instructor and input into *SIPLeS-II.* The equivalent component pairs, the context-based equivalent component pairs, and the nonequivalent pairs are also accumulated by the variation-learning process when the additional model programs are used to diagnose the set of student programs.

(2) *Online diagnosis of student programs*: With the model programs and component pairs accumulated, *SIPLeS-II* is then used online in a programming tutoring system to diagnose student programs submitted for TaskA.

The main issues studied in our empirical evaluation include:

(1) Number of model programs that are required in the step of *online diagnosis of student programs* for the system to achieve a satisfactory diagnostic performance (e.g., more than 95% of the student programs can be diagnosed correctly).

(2) Number of student programs that are required in the step of *model program accumulation and variation learning* for the system to accumulate sufficient model programs to achieve the stated diagnosis criterion.

(3) A comparison between the diagnostic performance of *SIPLeS-II* and the diagnostic performance of a tutor.

12.2 Programming tasks and student programs used in the evaluation

SIPLeS-II has been tested by using it to diagnose 525 student programs for nine different programming tasks as stated below. The task description for *Task1* was given in Section 3. Task descriptions for *Task2* to *Task9* are given in Appendix A.

(1) *Task1*: numerical calculation—TaxiFee

(2) *Task2*: class-header definition—RectangleClassDefinition

(3) *Task3*: rectangle object initialization—RectangleInitialization

(4) *Task4*: drawing a rectangle—RectangleDraw

(5) *Task5*: square object initialization— SquareInitialization

(6) *Task6*: drawing a square—SquareDraw

(7) *Task7*: rectangle object initialization by making use of inheritance— RectangleInitialization1

(8) *Task8*: drawing a rectangle by making use of inheritance—RectangleDraw1

(9) *Task9*: drawing a square and a rectangle by making use of polymorphism—TestMethod

Student programs were collected from two sources. First, student programs for the first programming task were gathered using the Internet. We created a web site containing seven Smalltalk/Java programming exercises, enabling students to write and submit their Smalltalk/Java programs. Second, we collected the remaining student programs from 62 third-year university students who were studying a course entitled “Object-oriented Methods” taught by us.

The number of statements in a typical student program ranged from 4 to 20 statements. An appreciation of the size and the complexity of the student programs can be obtained by studying Appendix A.

Of the 525 student programs used in the empirical evaluation, many programs were incorrect and contained various errors. Table 4 shows the percentage of the incorrect programs.

Table 4

Number of incorrect student programs and errors in student programs

	Task1	*Task2*	*Task3*	*Task4*	*Task5*	*Task6*	*Task7*	*Task8*	*Task9*	*Average*
Total no. of SPs	37	61	61	61	61	61	61	61	61	58
No. of incorrect SPs	16	11	9	32	30	47	44	54	39	31.33
Percentage of incorrect SPs	43%	18%	15%	52%	49%	77%	72%	89%	64%	53%
No. of errors	23	11	13	106	83	262	84	165	12	84.33
No. of errors per incorrect SP	1.44	1.00	1.44	3.31	2.77	5.57	1.91	3.06	0.31	2.69

Table 4 shows that, on average, 53% of the student programs used in the evaluation were incorrect student programs. On average, an incorrect student program contained 2.69 errors. The table also shows that if an incorrect student program was for a more complicated task, such as *Task4* or *Task6*, more errors might appear in an incorrect student program.

12.3 Evaluation method

The empirical evaluation was conducted in two steps, called *Step1* and *Step2,* corresponding to the two steps in the use scenario described in Section 12.1. For every programming task, we randomly divided the student programs (SPs) into two nearly equal sets, *Set1* and *Set2,* which are used in *Step1* and *Step2* respectively. Table 5 shows the details of the numbers of student programs for each programming task used in *Step1* and *Step2.*

Table 5

Number of student programs used in *Step1*and *Step2*

	Task1	*Task2*	*Task3*	*Task4*	*Task5*	*Task6*	*Task7*	*Task8*	*Task9*	*Total*
No. of SPs in *Set1*	18	30	30	30	30	30	30	30	30	258
No. of incorrect SPs in *Set1*	8	10	8	17	15	26	27	29	22	162
No. of errors in *Set1*	11	10	12	54	40	145	46	85	8	411
No. of SPs in *Set2*	19	31	31	31	31	31	31	31	31	267
No. of incorrect SPs in *Set2*	8	1	1	15	15	21	17	25	17	120
No. of errors in *Set2*	12	1	1	52	43	117	38	80	4	348

12.4 Evaluation results

Number of model programs

The number of model programs accumulated in *Step1* is shown in Table 6 together with the number of student programs used in *Step1.*

Table 6

Number of model programs learned in *Step1*

	Task1	*Task2*	*Task3*	*Task4*	*Task5*	*Task6*	*Task7*	*Task8*	*Task9*	*Average*
No. of SPs in *Set1*	18	30	30	30	30	30	30	30	30	28.67
No. of MPs learned	4	1	1	1	2	2	2	3	2	2.00

Table 6 shows that the number of model programs learned in *Step1* ranged from 1 to 4 for different programming tasks. For programming tasks such as a class-header definition (e.g., *Task2*), only one model program was sufficient. For programming tasks such as a method definition (e.g., *Task1*), up to four model programs were used. The four model programs for *Task1* are indicated in Appendix A. On average, two model programs were accumulated for a programming task during *Step1.* From the data in Table 6, we see that in transformation-based diagnosis, although the number of model programs needed for perfect online diagnosis is theoretically undecidable, the number of model programs needed in real application is small.

Rate of successful diagnosis

The number of student programs diagnosed correctly by *SIPLeS-II* in *Step2* is shown in Table 7 together with the number of student programs used in *Step2.*

Table 7

Number of student programs diagnosed correctly by *SIPLeS-II* in *Step2*

	Task1	*Task2*	*Task3*	*Task4*	*Task5*	*Task6*	*Task7*	*Task8*	*Task9*	*Average*
No. of SPs in *Set2*	19	31	31	31	31	31	31	31	31	29.67
No. of SPs diagnosed correctly	19	31	31	29	31	30	31	31	31	29.33
Rate of correct diagnosis	100%	100%	100%	93.55%	100%	96.77%	100%	100%	100%	98.88%
No. of SPs diagnosed incorrectly due to the lack of model programs	0	0	0	2	0	1	0	0	0	0.33
No. of SPs diagnosed incorrectly due to missing errors	0	0	0	0	0	0	0	0	0	0

Table 7 shows that, with the model programs learned in *Step1*, *SIPLeS-II* could diagnose most of the student programs correctly. The rate of successful program diagnosis by *SIPLeS-II* was 98.88%. *SIPLeS-II* incorrectly diagnosed three programs (two for *Task4* and one for *Task6*) in *Step2* due to the lack of one model program for each of *Task4* and *Task6*. It could not recognize Program45 (a correct student program) and Program58 (an incorrect student program) for *Task4* because both programs used a new algorithm unknown to the system. It also could not recognize Program36 (a correct student program) for *Task6* because the student program used a new algorithm.

The execution performance of *SIPLeS-II* is satisfactory. In empirical evaluation, we used a PC with a CPU speed of 120Mhz. It diagnosed 62 student programs in 45 seconds including writing diagnosis reports to a hard disk. This means that it took less than 1 second to diagnose a student program.

Comparing *SIPLeS-II* with a tutor

In addition to the diagnosis performed by *SIPLeS-II*, we also asked a tutor to assess all the student programs in *Set2.* The assessment results reported by the tutor are shown in Table 8.

Table 8

Number of student programs diagnosed correctly by tutor in *Step2*

	Task1	*Task2*	*Task3*	*Task7*	*Task5*	*Task6*	*Task4*	*Task8*	*Task9*	*Average*
No. of SPs in *Set2*	19	31	31	31	31	31	31	31	31	29.67
No. of SPs diagnosed correctly	15	31	31	17	31	30	31	22	30	26.44
Rate of correct diagnosis	78.95%	100%	100%	54.84%	100%	96.77%	100%	70.97%	96.77%	89.14%
No. of SPs diagnosed incorrectly due to the lack of model programs	0	0	0	0	0	0	0	0	0	0
No. of SPs diagnosed incorrectly due to missing errors	4	0	0	14	0	1	0	9	1	3.22

We compared the assessment results reported by the tutor (in Table 8) with the results provided by *SIPLeS-II* (in Table 7). The comparison shows that the tutor incorrectly assessed 29 student programs (out of a total of 267) due to missing errors. The rate of successful assessment by the tutor was 89.14%, lower than the rate of successful diagnosis by *SIPLeS-II*, which was 98.88%. Therefore, *SIPLeS-II* performed about 11% better than a human tutor did in diagnosing student programs.

The comparison revealed that the tutor diagnosed student programs incorrectly due to missing errors by carelessness. The human tutor performs better in being able to recognize different algorithms used by the students in their programs. However, *SIPLeS-II*, as an automatic diagnosis system, performs better overall. Its diagnostic weakness arose from the lack of model programs.

13. Discussion, summary and future work

In this paper, we proposed transformation-based diagnosis, a new appraoch to automatic diagnosis of students' programming errors for use in programming tutoring systems. In transformation-based diagnosis, automatic diagnosis of student programs is achieved by comparing the student program with the model program(s) at the semantic level after both have been standardized by program transformations.

Various program representations including AST and AOPDG are used. For pure object-oriented programming languages such as Smalltalk, a class definition uses the same form of message sending. Therefore, the program representations discussed in this paper are applicable not only to method definitions but also to class definitions. The AOPDG representation combines the strengths of McGregor's OPDG representation with Yang's PRG representation. The flow graph creation in AOPDG is improved in order to make it not only applicable to object-oriented programming languages, including C++ and Java, but also to pure object-oriented programming languages such as Smalltalk.

Program transformation has been extensively studied in the fields of compiler design [20], automatic logic and functional program generation [22], software maintenance [15], and parallel program optimization [20]. However, studies on program transformations for program standardization are rare. A method to remove variations in programs is proposed in [10]. However, the representation used in the method does not support program dependence graph and semantic level program comparison. Furthermore, some programs resulted from the transformations are difficult to be understood by the students. In transformation-based diagnosis, the AOPDG program representation supports the semantic level program comparison. The transformations used in our approach are only those applicable at source code level [18], not those applicable at intermediate code level, and are only those necessary for handling the possible semantics-preserving program variations. Transformed programs are easy to be understood by the students.

In general, there are three levels of program comparisons: simple text level comparison, syntax level comparison [5], and semantic level comparison [11], [13], [27]. The comparison algorithm used in our approach is a semantic level comparison algorithm. It extends Yang's partitioning algorithm to the AOPDG representation.

All 13 possible semantics-preserving variations in student programs are handled by various strategies shown in Table 3. The programming errors in student programs are identified by the comparison and the error detection step. The diagnosis report not only indicates the correctness of the programs, but it also pinpoints the errors in the incorrect programs and provides corrections for the errors. The errors identified by our method include not only superficial semantic errors such as incorrect expressions, but also non-superficial semantic errors such as incorrect program logical structures.

Scalability is always a difficulty for an automatic program diagnosis system. *Source-to-specification* approaches require a more extensive goal/plan library in order to handle larger student programs. Theoretically, when transformation-based diagnosis deals with larger student programs, it might also face the difficulty of requiring a more extensive library of model programs.

In practice, however, scalability does not appear to be a problem. We counted the number of statements in 100 methods defined in classes in the Smalltalk development environment VisualWorks 2.5. The 100 methods were selected randomly. We found that a method

definition contains a minimum one statement and a maximum 19 statements. On average, a method definition has six statements.

Furthermore, as described in Section 12.2, all student programs used in the empirical evaluation are real student programs. Student programs for *Task2* to *Task9*, which account for 93% of all the student programs, were coded by 62 third-year university students who were studying a programming course. Hence, the size of the student programs used in the empirical evaluation reflects the size of real student programs. Therefore, while scalability is a theoretical limitation, it appears not to be a problem in practice.

In summary, transformation-based diagnosis is a new approach, using techniques of program analysis and program matching, to the problem of automatic diagnosis of student programs. Although it is not a complete solution to the problem, compared to other existing approaches, it has the following noteworthy features.

(1) ***An approach to automatic diagnosis of student programs:*** It satisfies the requirements for automatic program diagnosis set out in Section 1. The correct diagnosis rate can be as high as 98.88% for student programs for various programming tasks.

(2) ***A convenient and feasible approach:*** The method requires only model programs as input to the diagnosis of programs. It is therefore *not* necessary to pre-study programming tasks and write program templates as well as program specifications for the programming tasks. The empirical evaluation further indicates that for satisfactory diagnosis of student programs, the number of model programs required is small.

(3) ***A more rigorous approach:*** Compared to other existing approaches, transformation-based diagnosis is more rigorous because the results of the program analysis, program transformation, and program comparison are provable.

(4) ***A safer approach:*** Compared to other existing approaches, transformation-based diagnosis is safer approach. It may regard an actually correct statement as an incorrect statement, but the approach will never regard an actually incorrect statement to be a correct statement.

(5) ***A general approach:*** The generality of program representations used in our approach makes the approach applicable to other object-oriented programming languages such as C++ and Java as well as non-objected-oriented programming languages such as Pascal and C. We have also completed a case study applying transformation-based diagnosis to Java programs. After we have represented Java programs in AST representation, which is based on a set of BNFs for the Java language, which is similar to that for the Smalltalk language, the remaining processes in transformation-based diagnosis are equally applicable to Java programs nearly without any change required. Most of the basic program transformations applied on Smalltalk programs are also applicable to Java programs. Some basic transformations specific to Smalltalk programs, such as s*tatement separation,* must be changed to transformations that are specific to Java programs, such as x ? y : z → if (x) {y} else {z}.

In this study, we also developed two techniques: a method to standardize programs and a method to match programs rigorously at the semantic level. Program standardization is a field that has not been investigated before.

Transformation-based diagnosis also establishes a framework in which a student program is "recognized" by matching it against a model program after both programs are standardized. This framework as well as the techniques of program standardization and program matching may also be useful for programming understanding and software maintenance.

We re-itemize the limitations of transformation-based diagnosis. They include the following: (1) The intentions of the student programs are not represented in transformation-based diagnosis. (2) Variations at the *system* level, such as different design of class hierarchies, and *inter-procedural* variations, are not handled in transformation-based diagnosis.

Our current and planned future work includes the following: (1) We are refining the method of transformation-based diagnosis. (2) We also plan to carry out further study in the field of programming language design and implementation based on the implications derived from this study. We are interested to know whether it is possible to determine a set of principles under which a programming language can be designed with merits in respects of program standardization, automatic program understanding, and automatic program generation.

Appendix A Programming task descriptions and model programs

Programming task descriptions and model programs used in the empirical evaluation are given in this appendix. The nine programming tasks used in the empirical evaluation are:

(1) *Task1*: numerical calculation—TaxiFee

(2) *Task2*: class-head definition—RectangleClassDefinition

(3) *Task3*: rectangle object initialization—RectangleInitialization

(4) *Task4*: drawing a rectangle—RectangleDraw

(5) *Task5*: square object initialization— SquareInitialization

(6) *Task6*: drawing a square—SquareDraw

(7) *Task7*: rectangle object initialization by making use of inheritance— RectangleInitialization1

(8) *Task8*: drawing a rectangle by making use of inheritance—RectangleDraw1

(9) *Task9*: drawing a square and a rectangle by making use of polymorphism—TestMethod

The task description for *Task1* was given in Section 3. In Section A.1, we show all four model programs used in the diagnosis of the student programs in the empirical evaluation. In Section A.2, we give the task descriptions for *Task2* through *Task9* and one model program for each of these tasks.

A.1 Model programs for *Task1*

First model program for *Task1:*

```
taxiFeeWith: mile isBookingCase: bookingCase
| orderFee price |
bookingFee := 0.0.
price := 2.5.
bookingCase
      ifTrue: [
            bookingFee := 2.0.
            price := 3.0].
```

```
^price * mile + bookingFee
```

Second model program for *Task1:*

```
taxiFeeWith: mile isBookingCase: bookingCase
bookingCase
    ifTrue:
        [^(2.0 + (mile*3.0))]
    ifFalse:
        [^(0.0 + (mile*2.5))]
```

Third model program for *Task1:*

```
taxiFeeWith: mile isBookingCase: bookingCase
| a |
a := mile * 2.5.
bookingCase
    ifTrue:
        [ a := mile*3.0+2.0].
^a
```

Fourth model program for *Task1:*

```
taxiFeeWith: mile isBookingCase: bookingCase
^(bookingCase) ifTrue: [ mile*3.0 +2.0] ifFalse: [mile*2.5]
```

A.2 Task descriptions and model programs for *Task2* through *Task9*

Task descriptions for *Task2* through *Task9:*

```
Two classes, Square and Test, are defined below. The Test class is used to
test the behavior of the Square class. An object of the Square class
instantiated with a length of 5, a border of '*' and an interior of ' '
produces the following output:
*****
*   *
*   *
*   *
*****

Object subclass: #Square
    instanceVariableNames: 'border interior length '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Shapes'

initializeWith: l with: b with: i
    length := l.
    border := b.
    interior := i.
    ^self

draw
    | s |
    s := ''.
    1 to: length do: [ :i |
        1 to: length do: [ :j |
        ((i=1) | (i=length) | (j =1) | (j=length))
            ifTrue: [ s := s, border ]
```

```
                ifFalse: [ s := s, interior ].
            ].
            s := s, '\'.
    ].
    DialogView warn: s withCRs.
    ^self

Object subclass: #Test
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Shapes'

testMethod
    "This is a class method. Usage: Test testMethod"
    | s |
    s  := Square new.
    s initializeWith: 5 with: '*' with: ' '.
    s draw.
    ^self
```

Step 1

Modify the above Square class and re-produce a Rectangle class below. The expected output from the code is as follow.

```
********
*       *
*       *
********
```

A model program for *Task2:*

```
Object subclass: #Rectangle
    instanceVariableNames: 'border interior length width '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Shapes'
```

A model program for *Task3:*

```
initializeWith: w with: l with: b with: i
    width := w.
    length := l.
    border := b.
    interior := i.
    ^self
```

A model program for *Task4:*

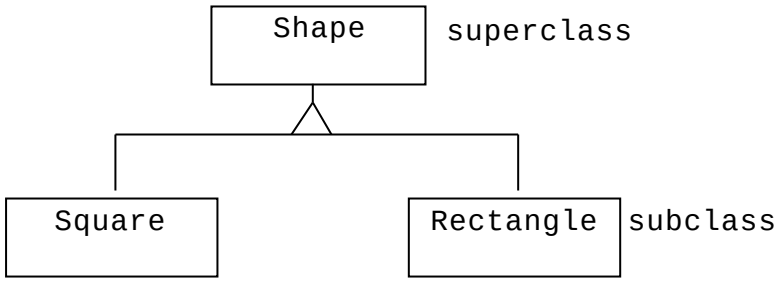
```
draw
    | s |
    s := ''.
    1 to: length do: [ :i |
        1 to: width do: [ :j |
        ((i=1) | (i=length) | (j =1) | (j=width))
            ifTrue: [ s := s, border ]
            ifFalse: [ s := s, interior ].
        ].
        s := s, '\'.
    ].
    DialogView warn: s withCRs.
```

```
    ^self
```

The task descriptions for *Task2* through *Task9* continue:

Step 2

Assuming the following class hierarchy:



A partial code for drawing shapes is given below. The definition of Shape and Test class is given. Class heads of Square and Rectangle are also given. You are required to complete the code for the:

(1) Square initialization method
(2) Square **draw** method
(3) Rectangle initialization method
(4) Rectangle **draw** method

The expected output is given here:

```
*****
*   *
*   *
*   *
*****
########
#      #
#      #
#      #
########
```

Object subclass: #Shape
    instanceVariableNames: 'border interior length '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Shapes'

initializeWith: l with: b with: i
    length := l.
    border := b.
    interior := i.
    ^self

border
    ^border

interior
    ^interior

len
    ^length

```
draw
	"To be implemented by subclasses."
	^self

Object subclass: #Test
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Shapes'

testMethod
	"This is a class method. Usage: Test testMethod"
	| s r |
	s := Square new.
	s initializeWith: 5 with: '*' with: ' '.
	r  := MyRectangle new.
	r initializeWith: 8 with: 5 with: '#' with: ' '.
	s draw.
	r draw.
	^self

Shape subclass: #Square
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Shapes'
```

A model program for *Task5*:

```
initializeWith: l with: b with: i
	super initializeWith: l with: b with: i.
	^self
```

A model program for *Task6*:

```
draw
	| s |
	s := ''.
	1 to: self len do: [ :i |
		1 to: self len do: [ :j |
		((i=1) | (i=self len) | (j =1) | (j=self len))
			ifTrue: [ s := s, self border ]
			ifFalse: [ s := s, self interior ].
		].
		s := s, '\'.
	].
	DialogView warn: s withCRs.
	^self
```

The task descriptions for *Task2* to *Task9* continue:

```
Shape subclass: #Rectangle
	instanceVariableNames: 'width '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Shapes'
```

A model program for *Task7*:

```
initializeWith: w with: l with: b with: i
	super initializeWith: l with: b with: i.
	width := w.
```

```
    ^self
```

A model program for *Task8*:

```
draw
    | s |
    s := ''.
    1 to: self len do: [ :i |
        1 to: width do: [ :j |
        ((i=1) | (i=self len) | (j =1) | (j=width))
            ifTrue: [ s := s, self border ]
            ifFalse: [ s := s, self interior ].
        ].
        s := s, '\'.
    ].
    DialogView warn: s withCRs.
    ^self
```

The task descriptions for *Task2* to *Task9* continue:

```
Step 3

Modify the testMethod of the Test class in Step 2 to demonstrate
polymorphism.
```

A model program for *Task9*:

```
testMethod
    "This is a class method. Usage: Test testMethod"
    | s r l |
    s := Square new.
    s initializeWith: 5 with: '*' with: ' '.
    r  := Rectangle new.
    r initializeWith: 8 with: 5 with: '#' with: ' '.
    l := List new: 2.
    l at: 1 put: s; at: 2 put: r.
    l do: [ :item | item draw ].
    ^self
```

References

[1] D. Allemang, "Using Functional Modes in Automatic Debugging," *IEEE Expert*, Vol. 6 no. 6, pp.13-18, 1991.

[2] J. R. Anderson, R. Farrell, R. Sauers, "Learning to Program in Lisp," *Cognitive Science*, Vol. 8, pp. 87-129, 1984.

[3] S. K. Abd-El-Hafiz, and V.R. Basili, *A Knowledge-based Approach to Program Understanding*. Massachusetts: Kluwer Academic Publishers, 1995.

[4] A. Adam, and J. Laurent, "A System to Debug Student Programs," *Artificial Intelligence*," Vol. 15, no. 1, pp. 75-122, 1980.

[5] M. Elsom-Cook, and B. du Boulay, "A Pascal Program Checker", J. Self, ed., *Artificial Intelligence and Human Learning*. Chapman and Hall, pp. 361-373, 1988.

[6] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages*, Vol. 9, no. 3, pp. 319-349, 1987.

[7] T. S. Gegg-Harrison, "Exploiting Program Schemata in an Automated Program Debugger," *Journal of Artificial Intelligence in Education*, Vol. 5, no. 2, pp. 255-278, 1994.

[8] A. Goldberg, and D. Robson, *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[9] J. S. Hahn and J. Kim, "Automatic Problem Description from Model Program for Knowledge-based Program Tutor," *Automatic Software Engineering*, Vol. 4, pp. 439-461, 1997.

[10] N. Hattori and N. Ishii, "A Method to Remove Variations in Source Codes," *Information and Software Technology*, Vol. 38, pp. 25-36, 1996.

[11] S. Horwitz, "Identifying the Semantic and Textual Differences between Two Versions of a Program," *ACM SIGPLAN Notices*, Vol. 25, no. 6, pp. 234-245, 1990.

[12] J. E. Hopcroft, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing, 1979.

[13] D. Jackson and D. A. Ladd, "Semantic Diff: a Tool for Summarizing the Effects of Modifications," *Proceedings of International Conference on Software Maintenance,* pp. 243-252, 1994.

[14] W. L. Johnson, and E. Soloway, "Proust: Knowledge-based Program Understanding," *IEEE Transactions on Software Engineering*, Vol. SE-11, no. 3, pp. 11-19, 1985.

[15] W. Kozaczynski, J. Ning, and A. Engberts, "Program Concept Recognition and Transformation," *IEEE Transactions on Software Engineering*, Vol. 18, no. 12, pp. 1065-1075,1992.

[16] E. Lemut, B. du Boulay, and G. Dettori, *Cognitive Models and Intelligent Environments for Learning Programming*. Springer-Verlag, 1992.

[17] C. Looi, "Automatic Debugging of Prolog Programs in a Prolog Intelligent Teaching System," *Instructional Science*, Vol. 20, pp. 215-263, 1991.

[18] D. Loveman, "Program Improvement by Source to Source Transformation," *Journal of ACM*, Vol. 24, no. 1, pp. 121-145, 1977.

[19] J. D. McGregor, B. A. Malloy, and R. L. Siegmund, "A Comprehensive Program Representation of Object-oriented Software," *Annals of Software Engineering*, Vol. 2, pp. 51-91, 1996.

[20] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[21] W. R. Murray, *Automatic Program Debugging for Intelligent Tutoring Systems*. Morgan Kaufmann Publishers, 1988.

[22] A. Pettorossi, and M. Proietti, "Transformation of Logic Programs: Foundations and Techniques," *Journal of Logic Programming*, Vol. 19-20, pp. 261-320, 1994.

[23] C. Rich, and L.M. Wills, "Recognizing a Program's Design: a Graph-parsing Approach," *IEEE Software*, Vol. 1, pp. 82-89, 1990.

[24] E. Y. Shapiro, *Algorithmic Program Debugging.* Cambridge. MA: MIT Press, 1983.

[25] G. Thorburn, and G. Rowe, "PASS: An Automated System for Program Assessment," *Computers and Education,* Vol. 29, no. 4, pp. 195-206, 1997.

[26] H. Ueno, "Concepts and Methodologies for Knowledge-based Program Understanding-the ALPUS's Approach," *IEICE Transactions on Information and Systems*, Vol. E78-D, no. 9, pp. 1108-1117, 1995.

[27] W. Yang, S. Horwitz, and T. Reps, "A Program Integration Algorithm that Accommodates Semantics-preserving Transformations," *ACM Transactions on Software Engineering and Methodology*, Vol. 1, no. 3, pp. 310-354, 1992.

BIOGRAPHY OF EACH AUTHOR

Songwen Xu received the B.S. degree in electronics and the M.S. degree in computer science from Peking University, China, in 1988 and 1994 respectively, and the Ph.D. degree in computer science from National University of Singapore in 1999.

He is currently working at PeopleSoft, Inc. His research has been focused on intelligent tutoring systems, programming learning environments, and techniques for automatic program diagnosis, including program representation, program transformation, program comparison, and program matching.

Chee Yam San is an Associate Professor in the Department of Computer Science, School of Computing, National University of Singapore. Prof Chee is an editorial board member of the International Journal of Educational Telecommunications. He currently conducts research on learning environments and the learning sciences, focusing especially on the use of distributed multimedia computer technologies in promoting learning and educational goals.