Automatic Diagnosis of Student Programs in Programming Learning Environments

Songwen Xu and Yam San Chee
School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 119260
email: {xusw, cheeys}@comp.nus.edu.sg

Abstract

This paper describes a method to automate the diagnosis of students' programming errors in programming learning environments. In order to recognize correct students' programs as well as to identify errors in incorrect student programs, programs are represented using an improved dependence graph representation. The student program is compared with a specimen program (also called a model program) at the semantic level after both are standardized by program transformations. The method is implemented using Smalltalk in *SIPLeS-II,* an automatic program diagnosis system for Samlltalk programming learning environments. The system has been tested on approximately 330 student programs for various tasks. Experimental results show that, using the method, semantic errors in a student program can be identified rigorously and safely. Semantics-preserving variations in a student program can be eliminated or accommodated. The tests also show that the system can identify a wide range of errors as well as produce indications of the corrections needed. This method is essential for the development of programming learning environments. The techniques of the improved program dependence graph representation, program standardization by transformations, and semantic level program comparison are also useful in other research fields including program understanding and software maintenance.

1 Introduction

It is essential for a programming learning environment to be able to determine whether a program written by a student is coirect. The system should recognize correct student programs even they contain diverse variations. The system should also be able to identify errors together with the corrections needed if the student's program is incorrect.

Research on the automatic diagnosis of student programs relates to many fields such as programming learning environments [Ramadhan and du Boulay, 1992; Ueno,1995], automatic program assessment [Thorbum and Rowe, 1997], program analysis and understanding [Rich and Wills, 1990], and software maintenance [Kozaczynski *et al.,* 1992]. However, there are few prototypes of systems focusing on the problem of automatic diagnosis of programming errors. Existing work includes Adam & Laurent [1980], Johnson & Soloway [1985], and Murray [1988]. To our knowledge, no existing system performs the automatic diagnosis of programming errors entirely satisfactorily today due to the difficulty of the problem.

In this paper, we describe a method to automate the diagnosis of students' programming errors in programming learning environments. Specimen programs (also called model programs) are used as the input to the diagnosis of student programming errors. The automatic diagnosis of students' programming error, is achieved by semantically comparing the student program with a specimen program after both have been standardized by program transformations. Programs are represented as Abstract Syntax Trees (ASTs) and Augumented Object-oriented Program Dependence Graphs (AOPDGs).

The method has been implemented using Smalltalk in *SIPLeS~II,* an automatic program diagnosis system for Smalltalk programming learning environments. The system has been tested on approximately 330 student programs for various tasks. Experimental results show that, using the method, semantic errors in a student program can be identified rigorously and safely. Semantics-preserving variations in a student program can be either eliminated or accommodated.

This method is essential for the development of programming learning environments. The generality of the method makes it applicable to other object-oriented programming languages such as C++ and Java as well because the AOPDG representation is applicable to general object-oriented programming languages. It is also applicable to non-objected-oriented programming languages where programs can be represented in ordinary augmented dependence graphs. The techniques of (a) the improved program dependence graph representation, (b) program standardization by transformations, and (c) semantic level program comparison, are also useful in other research fields including program understanding and software maintenance.

2 Automatic Diagnosis Procedure

In our approach, for a programming task, different model programs are used to diagnose student programs that are coded based on different algorithms. Student programs using the same algorithm are standardized and compared to their corresponding model program. The automatic diagnosis procedure in *SIPLeS-II* is shown in Fig. 1.

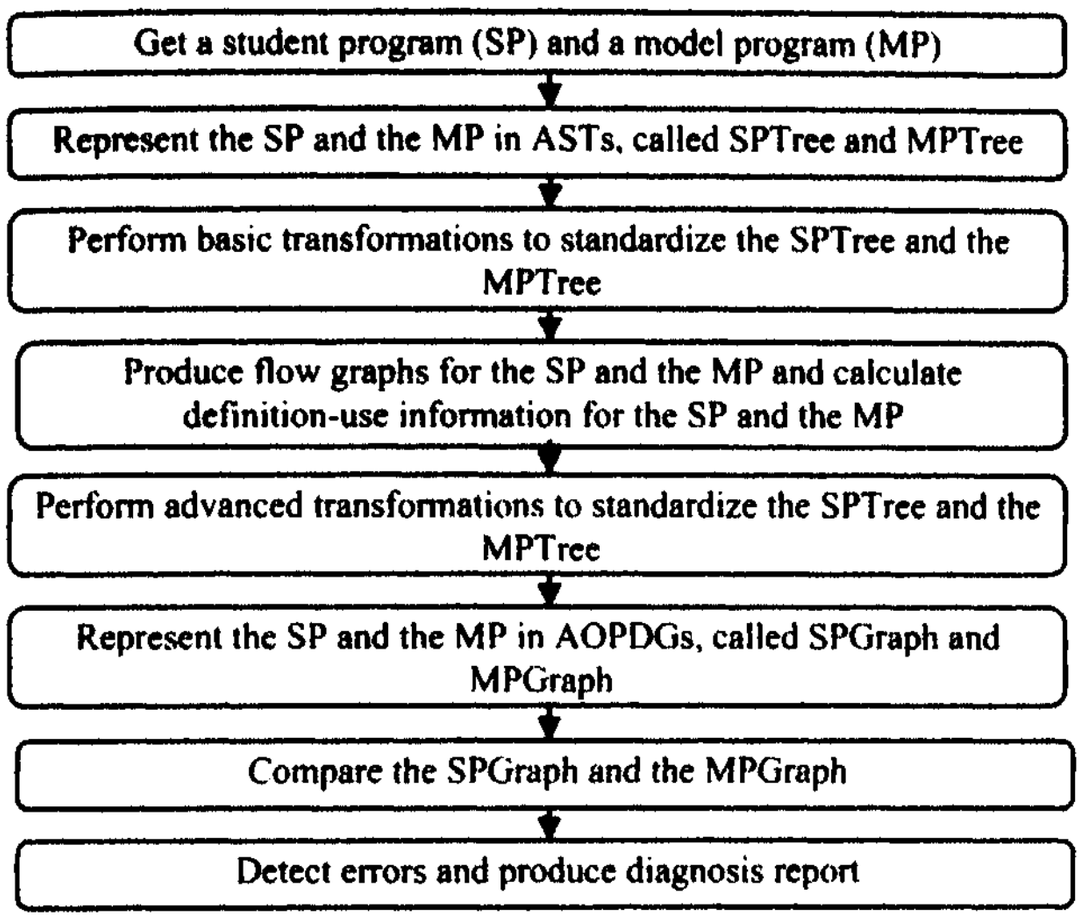

Fig. 1. Automatic diagnosis procedure in *SIPLeS-H*

In *SIPLeS-II,* a student program (SP) and a model program (MP) represented in source code are obtained. 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) for Smalltalk, the AST representations of the two programs are produced, called SPTree and MPTree. The AST representation is suitable for the program analysis at the syntax level and for the program transformations [Loveman, 1977]. After that, basic transformations, which do not require definition-use information (DU information) are performed to standardize the SPTree and the MPTree.

In order to calculate DU information, flow graphs for the student program and the model program are produced based on the SPTree and the MPTree. Several kinds of □ vertices are augmented to the flow graphs in order to combine them with features of Static Single Assignment (SSA) form where every use of a variable is only defined by one definition. The DU information for the two programs are calculated and used both in the advanced standardization transformations and the generation of the data dependence subgraphs in AOPDG representations. We call the AOPDGs for the SP and the MP SPGraph and MPGraph respectively.

The comparison between the SPGraph and the MPGraph produces the following results. (1) A mapping between SP statements and their semantics-equivalent MP statements. We call this the equivalent map. (2) A mapping between SP statements and MP statements that are semantics-equivalent but textually different from the SP statements. We called this the textual difference map. (3) A set that includes unmatched statements in the SP and the MP. The set is called the unmatched set.

Based on the comparison results, the SP statements in the equivalent map and the textual difference map are recognized as correct statements in the error-detection step. For every unmatched SP statement, the most similar unmatched MP statement is found and the differences between the two statements are identified. Among these differences, those that are actually legal variations are learned and eliminated by the system. Unresolved differences are reported as errors in the student program. Unmatched statements in the SP with no unresolved difference identified are reported as controlling errors in the diagnosis report.

3 An example

We use a running example to explain our approach. The task description is given below.

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.

A model program is the following, where names for the method head and statements are given at 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)
```

A possible student program is given below. The differences between the SP and the MP are indicated, they will be discussed later.

```
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 + bookingFee) * mile.
SS0013End^payment
```

Difference1
Difference2
Difference3
Difference4
Difference5
Difference6

3.1 Possible semantics-preserving variations in a student program

In programming learning environments, a program written by a student may have many semantics-preserving variations (SPVs) compared to a model program. In general, there are 12 possible types of variations. These are given below.

- SPV1: Different algorithms may be used. The student program will be quite different from the model program.
- SPV2: Different format of writing program at the source code level. For example, more or less spaces, comments, etc.
- SPV3: Different ways of writing message sequences. For example, message sequences may be written in cascaded messages.
- SPV4: Different temporary variable declarations. Temporary variables may be declared in a method temporary variable declaration or in a block temporary variable declaration.
- SPV5: Different ways of writing algebraic expressions.
- SPV6: Different messages used for a same control structure.
- SPV7: Different numbers of temporary variables or block temporary variables are used.
- SPV8: There are may be some dead codes or some statements for debugging purpose in the student program.
- SPV9: Different statement orders.
- SPV10: Different parameter names in the method head or different temporary variable names or different block temporary variable names are used.
- SPV11: Different control structure used. The way of computation in the SP is different from that in the MP.
- SPV12: Different ways of writing a statement. A component in a statement in SP is different from that in a statement in MP although the computational results of the two components are the same.

3.2 Differences between the MP and the SP

A human tutor may identify the following six differences between the student program and the model program.

- Difference 1 (SPV 10): Different parameter names are used in S Entry and SS0011.
- Difference2 (SPV 9): Different statement orders of 551121 and S1122 compared to that of MS1321 and MS1322.
- Difference3 (SPV 2): Different source-code format in 551122 compared to that in MS1322. There is an extra ""before the "J" in SS1122.
- Difference4 (SPV 11): Different values produced at the point of SS0011. price and bookingFee are calculated out of the if False: control structure in the MP, whereas they are calculated inside the if False: control structure in the SP. However, this is a semantics-preserving variation, because at the point of SS0012, the values produced are the same compared to the values produced in the MP at the point of MS0014End.
- Difference5 (Error): Different computation carried out in SS0012 compared to MS0014End. This is a semantic variation, which is an error in the student program.
- Difference6 (SPV 7): Different numbers of temporary variables are used in SS0012 and SS0013End. The difference changes the way computations are executed without changing the values computed.

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

4 Program Representations

Three representations of programs are used in the diagnosis procedure: source code representation, AST representation, and AOPDG representation. The AST representation is amenable for program transformation. The AOPDG representation combines the strengths of OPDG representation [McGregor *et al.,* 1996] and the program representation graph (PRG) [Yang *et al. ,* 1992], It is used for the semantic level comparison of programs [Horwitz,, 1990]. The merits of the AOPDG representation include: (1) it eliminates many variations existing at the source code level and AST level, (2) it is indispensable for the program comparison algorithm, and (3) it makes the comparison accommodate semantics-preserving behavioral changes in the student program compared to the model program.

4.1 AST representation

The generation of the abstract syntax tree for a program is based on the Backus-Naur Forms (BNF) of the programming language. An AST representation of a program is a frame-based representation based on the parsing tree with additional program analysis information added.

Representing programs in AST eliminates SPV2 in programs. In the running example, Difference3, the "." before the "]", is eliminated.

4.2 AOPDG representation

An AOPDG for a program is constructed based on the augmented object-oriented flow graph (AOFG) of the program and the DU information calculated. An AOFG is constructed from the OFG of the program, and an OFG is an improved flow graph. It accommodates the situation in pure object-oriented programs that statement sequence may appear in a block as a component of another statement. This is not allowed in programs such as C++ and Java. In the OFG, there

are additional flow edges coming out from a conditional node besides the usual true/false branches.

To construct an AOFG from an OFG, a vertex labeled "□initial: x := initialM" is added at the beginning of the OFG for each variable x that may be used before being defined. A □ vertex labeled "□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 " □exit x := x" is added immediately after the loop for each variable that is defined within the loop and is live after the loop.

An AOPDG consists of an augmented object-oriented control dependence subgraph (AOCDS) and an augmented object-oriented data dependence subgraph (AODDS). Similar to the construction of CDS and DDS, AOCDS for a program is constructed based on AOFG, and AODDS is constructed by calculating DU information on AOFG.

In AOCDS representation, there are six types of vertices: entry vertex, end vertex, statement vertex, initialization □ vertex, enter □ vertex, and exit □ 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: controltrue, controlfalse, controlloop, entertrue, and enterfalse, and the types of dependence edges are ftowl, flow2,..., flown, flowtrue, flowtalse, fbwenter, flowexit, and flowwhile.

By representing the programs in AOPDGs, SPV9 in programs is eliminated. In the running example, the AOPDG representations of the SP, SPGraph, are given in Fig. 2. Difference2 in the SP is eliminated by representing the SP and the MP in AOPDGs.

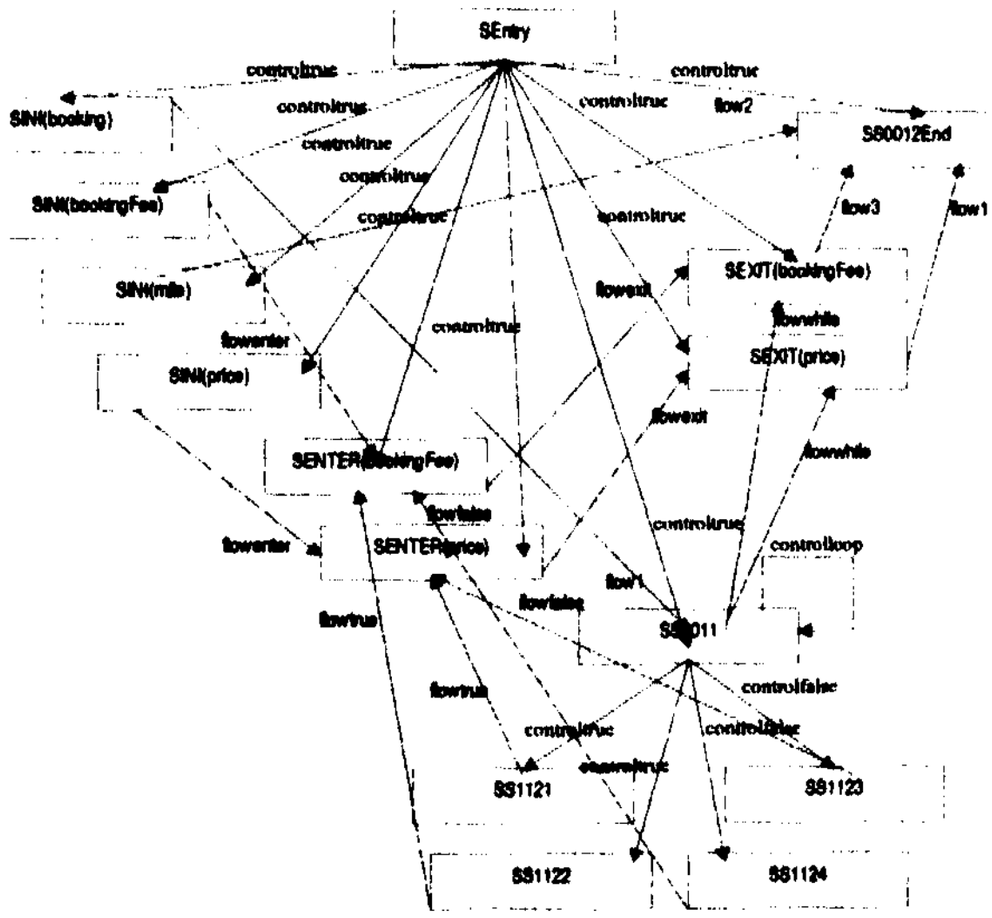

Fig. 2.The AOPDG representation of the student program

5 Program Transformations

The basic standardization transformations performed are as follows. (1) The statement separation standardizes cascaded messages to a message sequence. (2) The temporary declaration standardization makes all temporary variables to be only defined in the method temporary variable declaration. (3) The algebraic expression standardization applies a set of rules of associativity, commutativity, and distributivity on an algebraic expression until no more transformation rules can be applied. (4) Control structure standardization standardizes all control structures into one of three structures—ifTrue:ifFalse:, whileTrue:, and to:by:do:—by applying 14 transformation rules such as receiver ifTrue: b1->receiver ifTrue: b1 if False: 0, where b1 is a block.

With basic standardization transformations, SPV3, SPV4, SPV5, and SPV6 are eliminated in programs. In the running example, algebraic expression standardization is applied on both SPTree and MPTree, and control structure standardization is applied on the MPTree.

Advanced standardization transformations of forward substitution and dead code removal [Muchnick, 1997] are used to eliminate SPV7. In the running example, no advanced standardization transformation is applicable to the MP. For the SP, SS0013End is changed by forward substitution and SS0012 is removed by dead code removal. Difference6 in the SP is eliminated. The SP standardized by advanced standardization transformations is given below; this corresponds to the AOPDG in Fig.2.

```
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)
```

6 Program Comparison

The comparison algorithm is based on the idea that two statements with different operators, different operands, or different controlling predicates will have difference behaviors. The vertices in the student AOPDG (i.e. the statements in the student program) and the vertices in the model AOPDG are classified into a same partition set in initial partition. A stable coarsest refinement of the initial partition is computed using a basic partitioning algorithm [Yang *et* a/., 1992]. 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. The results of the comparison reveal the semantic differences between the student program and the

model program and accommodate the semantics-preserving behavior changes between the two programs.

SPVI1 is accommodated in program comparison. In the running example, the refined partition is given below, where the names of the partition sets are shown on left. The equivalent map, the textual difference map, and the unmatched map are also given.

Refined partition

{taxiFeeWith:isBookingCase:→ (MEntry SEntry)
INI → (all initialization ϕ vertices the SP and the 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)}
where NEW1, NEW2, and NEW3 are separated from ENTER, EXIT, and + respectively during the refinement.

The equivalent map

{taxiFeeWith:isBookingCase: 0.0 2.5 ifTrue:ifFalse: 3.0 2.0}

The textual difference map

{taxiFeeWith:isBookingCase: ifTrue:ifFalse:}

The unmatched map

{+ NEW3}

Difference 1 is correctly identified as textual difference in this step because taxiFeeWith:isBookingCase: ->(MEntry SEntry) and ifTrue :if False: -> (MSO013 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. Differences is correctly identified as a semantic error because +->(MS0014End) and NEW3->(SS0012End) are in the unmatched map.

7 Error Detection

The results of the comparison report those statements in the SP that have semantic errors. However, a programming learning environment should be able to pinpoint the errors in the incorrect statements and to provide corrections of the errors. An error detection step is necessary. In this step, SPV10 is eliminated by changing the model program according to the textual difference map. The system pinpoints errors in an incorrect student statement by comparing it with the most similar model statement. The system also learns equivalent expressions used in the SP and the MP to eliminate SPV12. The diagnosis report for the running example is given below.

```
==========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}
```

8 Variations and Handling Strategies

All of the 12 semantics-preserving variations except SPV1 are handled by various strategies discussed above. For SPV1, different model programs corresponding to different algorithms are used to diagnose student programs that use different algorithms. Hence, when a student program significantly differs from all the model programs, a new model program must be input by the teacher.

The semantic differences left in the student program are identified as student programming errors in the diagnosis report. It is possible that a reported error may actually be a semantics-preserving variation because it is not included in the 12 types of SPVs. However, in our approach, it is impossible to miss an error if one actually exists. Hence, the approach described here is both safe and conservative.

9 Results and Discussion

TABLE 1. Test results of *SIPLeS-II*

Task name	SPs	MPs	SPs diagnosed correctly	Rate of correct program diagnosis	Rate of correct statement diagnosis
TaxiFee	37	5	37	100%	100%
IsPerfectNumber	40	5	40	100%	100%
PickPalindrome	32	4	32	100%	100%
SquareClassDef	55	1	55	100%	100%
DrawSquare	55	3	55	100%	100%
RectangleClassDef	56	1	56	100%	100%
DrawRectangle	56	3	56	100%	100%

In this paper, we proposed a new approach for automatic diagnosis of students' programming errors in programming

learning environments. In this approach, automatic diagnosis of student programs is achieved by comparing the student program with the model program after both have been standardized by program transformations. The approach is implemented in a system called *SlPLeS-II* using Smalltalk/VisualWorks 2.5. It has been tested on approximately 330 student programs for 7 different programming tasks. The test results are shown in Table 1.

From Table 1, we see that both the rate of correct program diagnosis and the rate of correct statement diagnosis are 100% after the system learns sufficient model programs. From the figures shown in Table 1, the number of model programs needed for diagnosing a method definition is in the order of 3 to 5, and the number of the model programs needed for diagnosing a class definition is 1 only.

Our experimental data also show that after the system has processed about 25 student programs for a programming task, the possibility of failing to diagnose errors due to the lack of a model program is less than 10%. This means that the number of model programs required becomes quite stable after about 25 student programs are processed. It is reasonable to believe that, in practice, the number of model programs required is small although in theory the number of model programs needed is undecidable.

In summary, the new features of our approach are as follows:

- The AOPDG program representation reflects semantic information of the program, eliminates many non-semantic variations, and is amenable to transformations and comparison.
- Programs are analyzed, standardized, and compared rigorously at the semantic level. By "rigorously", we mean that the results are guaranteed to be correct.
- Student programming errors are identified safely. By "safely", we mean that the approach 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. It is a conservative approach.
- Correct programs are recognized by the system handling all the possible variations.
- Errors in incorrect programs are identified and corrections to the errors are also provided.

This method is essential for the development of programming learning environments. The techniques of (a)the improved program dependence graph representation—AOPDG, (b) program standardization by transformations, and (c) semantic level program comparison, are also useful in other research fields including program understanding and software maintenance.

Our future work includes refining the method and applying the approach to program understanding and software maintenance.

References

[Adam and Laurent, 1980] A. Adam, and J. Laurent. A System to Debug Student Programs. *Artificial Intelligence,* 15(1): 75-122, 1980.

[Ferrante *et al,* 1987] J. Ferrantc, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages,* 9(3): 319—349, 1987.

[Horwitz, 1990] S. Horwitz. Identifying the Semantic and Textual Differences between Two Versions of a Program. *ACM SIGPLAN Notices,* 25(6):234-245, 1990.

[Johnson and Soioway, 1985] W.L. Johnson, and E. Soloway. Proust: Knowledge-based Program Understanding. *IEEE Transactions on Software Engineering,* SE-11(3): 11-19, 1985.

[Kozaczynski *et ai,* 1992] W. Kozaczynski, J. Ning, and A. Engberts. Program Concept Recognition and Transformation. *IEEE Transactions on Software Engineering,* 18(12): 1065-1075,1992.

[Loveman, 1977] D. Loveman. Program Improvement by Source to Source Transformation. *Journal of ACM,* 24(1): 121-145, 1977.

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

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

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

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

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

[Thorbum and Rowe, 1997] G. Thorburn, and G. Rowe. PASS: An Automated System for Program Assessment. *Computers and Education,* 29(4): 195-206, 1997.

[Ueno, 1995] H. Ueno. Concepts and Methodologies for Knowledge-based Program Understanding-the ALPUS's Approach. *IEICE Transactions on Information and Systems,* E78-D(9):l 108-1117,1995.

[Yang *et al,* 1992] W. Yang, S. Horwitz, and T. Reps. A Program Integration Algorithm that Accommodates Semantics-preserving Transformations. *ACM Transactions on Software Engineering and Methodology,* l(3):310-354, 1992.