Model based Diagnosis of Student Programs

Leliane Nunes de Barros and Karina Valdivia Delgado

Abstract. An automatic program debugger, based on the AI model based diagnosis technique, can guide students to construct programming knowledge. During the hypotheses discrimination, the diagnosis system may allow students to acquire, better and faster, the necessary programming skills. In this paper we show the results of some experiments realized on a set of faulty student’s programs using ProPAT\_DEBUG, a MBD system specially constructed be used on an introductory programming course.

1 Introduction

Model based diagnosis (MBD), also called diagnosis from the first principles, can be described as the interaction between observations and predictions [2]. MBD has been typically applied to troubleshooting physical device. On one hand, we have the actual device whose behavior can be observed, on the other hand we have a model of that device (system description) which is used to make predictions about its behavior. Such model typically describes the components of the system, their connections and the behavior of the components. The difference between an observation and a prediction is called a discrepancy. A match between an observation and a prediction is called a corroborations. Both discrepancies and corroborations are used to identify which parts of the device are possibly incorrect [2].

While engineers troubleshoot mechanical or electrical systems to find broken parts by trying to understand the differences between the physical systems and their models, a computer programming teacher tries to understand the differences between the student program code and his intentions (or the problem goals). If we see the student program as the system the teacher wants to troubleshoot, it is interesting to notice that he does not have the program correct model to reason about, once there is a huge number of different solutions for a single computer problem. Instead, the teacher reasons about program fragment models with the semantics of each expression and sentence being given by the programming language. Further, when the teacher tries to really understand the student program intentions, he also reasons in terms of general and well-known programming strategies, trying to identify them into the student code and to check how well they were instantiated to solve the current problem.

An original application of Model Based Diagnosis techniques for program debugging was proposed by Franz Wotawa and Markus Stumptner in an Intelligent Debugging System of the Project Jade, to help advanced programmers to find bugs [12]. In the program model, language expressions and sentences are represented as components; the information flow is represented as a connection; and the components behavior are described based on the language semantics. The diagnostic solution is a set of possible faulty components, in this case, expressions or sentences of the program code with bugs.

Although the Jade Project [13] has achieved some interesting results, we believe that it can not be successfully applied to students. While in the Jade Project it is expected that a programmer will be able to easily correct his bugs by simple looking at the most probably failing parts of his program, a student will find this task very difficult and will probably not be able to learn from that. A step forward to promote such learning process, is to explore the phase where the students makes his predictions about the variable values to be done, in a more comprehensive way. This step goes in the same direction of a teacher trying to understand the student intentions. This work makes the assumption that this can be done by the use of knowledge about elementary programming strategies, also called elementary patterns [16], to establish a better communication between the diagnosis system and the student.

In this paper we describe the development of an automatic debugging system, named ProPAT\_DEBUG, which is part of a programming learning environment based on elementary patterns, called ProPAT (Section 1.1) [7]. We also show the results of some experiments realized on a set of student’s programs, including faults classified according to a taxonomy of typical student faults.

1.1 The ProPAT Plug-in

Research on cognitive theories about programming learning has shown evidences that experienced programmers retrieve old experiences on problem solving that can be applied and adapted to a new problem [10]. On the other hand, a novice programmer does not have yet any real experiences, but only the primitive structures from the programming language. Inspired on these ideas, a strategy to teach how to program is to present small programming pieces, instead of leaving the student to program from scratch. That is the proposal of the elementary patterns community: a group of experienced educators engaged to recommend programming pieces for novices [16].

Elementary patterns [16] are solutions to common problems, described in a way to ease their reuse. Patterns are simple, synthetic and provide information about the context in which it can be applied. Patterns are available in the Web for C, C++ and Java [16], including: selection patterns [3], repetition patterns [1] and others [4].

ProPAT is a programming learning environment using elementary patterns, that has been built as an Eclipse plug-in [8]. ProPAT provides an IDE for a first Computer Science course. In this environment, the student can construct a solution by selecting and adding elementary programming patterns into the editor [7]. The ProPAT Eclipse plug-in has two main perspectives: the Student Perspective (Figure 1) where the student can choose programming exercises and develop solutions for them, through pattern selections or write his own code; and the Teacher Perspective used by the teacher to specify new exercises and patterns that will be available to the student through the Student Perspective. This project has been first developed

\footnote{University of São Paulo, Brazil, email: leliane,kvd @ime.usp.br}
for the C programming language but we are currently developing a Java version of ProPAT.

Figure 1. ProPAT Student Perspective - Patterns View and Pattern Info View

The rest of this paper is organized as follows. Section 2 gives the background knowledge on Model Based Diagnosis. In Section 3 we discuss how the general theory of MBD can be applied to diagnose student’s programs. In Subsection 3.2, we give some details about the implementation of the MBD diagnosis system for computer programs, named ProPATDEBUG. Section 4 presents the results of some experiments realized on a set of faulty student’s programs using ProPATDEBUG. Section 5 compares the ProPAT system with Intelligent Tutoring Systems for programming. In the Section 6 we draw the final conclusions.

2 Model Based Diagnosis Task

2.1 MBD: the conceptual model

Diagnosis reasoning can be conceived as performing three subtasks: (1) symptoms detection, where a symptom is defined as an observation that deviates from expectation; (2) hypotheses generation, where the possible causes, taking into account the initial observation, are generated; and (3) hypotheses discrimination, that discriminates the hypotheses set based on additional observations [2].

- **Symptoms Detection.** Symptoms are defined as abnormal observations of the system outputs, i.e., an output value that deviates from the expected value. If no symptoms is detected, the diagnosis task will not be performed.

- **Hypotheses Generation.** This task has two subtasks: find contributors and transform contributors to a hypotheses set.
  
  - **Find Contributors.** The contributor set (for the abnormality observations) is a set of components which contains at least one component incorrect (or faulty). The contributor set is also called a conflict set [5]. A possible approach to perform this task is by simulation. Simulation yields to an expectation (e) for an initial observation (Oinit). If the observation is abnormal, and a set of contributors (c) has been used for deriving the expectation (e), then we say that (c) is the contributor set of the initial abnormal observation (Oinit,ab).

  - **Transform Contributors to Hypotheses Set.** The conflict sets are transformed into a hypotheses set. Every hypothesis in the hypotheses set is an explanation of all initial observations [2] and all its elements are supposed to be faulty. This task can be implemented using the Reiter’s Algorithm [15], that constructs the hitting set tree (Section 2.3).

- **Hypotheses Discrimination of programs.** For each hypothesis generated, a set of new observations must be made that will be given as the new input to the hypotheses generation task. To decide in which order the hypotheses are going to be tested, it is necessary to have a fault estimate value derived for each component, e.g., from test case execution.

2.2 MBD: the formal model


**Definition 1** [15]: A system is defined to be a pair (SD,COMP), where SD is the system description and COMP is a finite set of constants denoting the collection of components of the system. The system description SD is comprised of a set of first-order logic sentences describing the functionality of the components within the system (behavioral model) and the connections between the components of the system (structural model).

**Definition 2** [15]: Given an observation, OBS, (SD,COMP,OBS) is a diagnosis problem for the system (SD, COMP) with observation OBS.

**Definition 3** [15]: A diagnosis Δ for the system (SD, COMP) is a minimal subset of COMP such that:

\[ SD \cup OBS \cup \{ AB(C) \mid C \in \Delta \} \cup \{ ¬ AB(C) \mid C \in COMP \setminus \Delta \} \]

is consistent. Where \( AB(C) \) means that the component C has an abnormal behavior, i.e., it is faulty.

**Definition 4** [15]: A contributors set for (SD, COMP, OBS) is a set \( CO \subseteq COMP \) such that:

\[ SD \cup OBS \cup \{ ¬ AB(C) \mid C \in CO \} \]

is inconsistent.

**Definition 5** [15]: A hitting set for a collection of sets C is a set

\[ H \subseteq \bigcup_{S \in C} S \]

such that \( \forall S \in C, H \cap S \neq \emptyset \), i.e., a hitting set is a set that intercepts all the sets of the collection C. A minimal hitting set is an hitting set such that none of its subsets is an hitting set. When the collection set C corresponds to the set of all contributors sets CO for a diagnosis problem (SD, COMP, OBS), the minimal hitting set is the simplest explanation for the observations. The next theorem shows a constructive form to find the hypotheses from the contributor collection set.

**Theorem 1** [15]: The set \( \Delta \subseteq COMP \) is a diagnosis for (SD,COMP,OBS) if and only if \( \Delta \) is an minimal hitting set for the collection of all contributor sets for the diagnosis problem (SD,COMP,OBS).

2.3 A brief review of Reiter’s original algorithm

Reiter [15] proposed a diagnosis algorithm (for faulty systems in general) that computes all minimal hitting sets for a family of components sets \( F \). The algorithm generates an acyclic graph in which nodes are labeled by sets and arcs are labeled by elements of the set. The idea is that for each node labeled by a set \( S \), the arcs leaving
from it are labeled by the elements of \( S \). Let \( H(n) \) denote the set formed by the labels of the path going from the root to node \( n \). Node \( n \) has to be labeled by a set \( S \) such that \( S \cap H(n) = \emptyset \). If no such set can be found, the node is labeled by @. The idea is that every path finishing at a node labeled by @ is a hitting set, since it intersects all possible labels for the nodes.

### 3 Model Based Diagnosis of Computer Programs

The basic idea for diagnosing programs, instead of diagnosing physical devices, is to derive a System Description directly from the student program and the programming language semantics. This model must represent components, connections and its behavior based on the actual student program behavior which reflects its errors. The observations are the incorrect outputs in different points of the original program. The predictions are not made by the system, but by observations are the incorrect outputs in different points of the original program code. The predictions are not made by the system, but by the student and therefore this is the situation where he must communicate his expected values for the variables and be able to understand what are the possible faults in his program.

There are two approaches that can be used for program modeling: a value-based model [12] and a dependency based model [17]. They are used to detect functional faults but they are not recommended for detecting structural faults [17]. Functional faults are all faults that result from the storage of an incorrect value of some variable, in at least one possible evaluation trace. In particular, these faults include the use of an incorrect operator or the use of incorrect literals (variables). Examples of functional bugs are: omitting an operator (e.g., writing \( i \) instead of \( i + 1 \); using the wrong operator (e.g., writing \( i + i \) instead of \( ++i \)) or the wrong variable (e.g., \( a[i] \) instead of \( a[j] \)); missing an initializing of variables; a wrong modification of a value stored in a variable; errors with loop index initializations; exit tests that lead to an erroneous value of a variable. Structural faults, on the other hand, are source code bugs, which alter the structure of the underlying program. For example: missing statements, statements out of order, superfluous statements or access to an incorrect variable. As we show in Section 4 the diagnosis system ProPAT_deBUG can detect functional faults and some particular structural faults.

#### 3.1 The Value-based Model

In the ProPAT_deBUG system we have chosen to use the value-based approach [12] to construct the program model SD. Since the value-based model can eliminate wrong diagnosis by using additional run-time information (the expected values of variables provided by the students), it achieves better results than the dependency based model in most cases [13] and helps the student to participate on the diagnosis process.

In the value-based model: expressions and statements are represented as components (structural model); the semantics of the expressions and statements are described by sets of logical sentences (behavioral model). Components are connected if there is a flow of information between the corresponding expressions and statements. An information flow between an assignment and another statement occurs, p.e., if the assignment changes the value of a variable that is accessed by other statement, and there is no assignment changing the same variable in between [12]. Thus, to obtain the structural model we make the following:

- all variables are mapped to connections and whenever a variable occurs in an expression, this connection is used to connect the corresponding components. Each time a variable is used in the assignment’s left side, a new connection is created and used for all components that use it until the variable is used again in an assignment’s left side;
- sentences, assignments, conditionals, while loops, return statements and expressions, are mapped into components.

#### 3.2 ProPAT_deBUG System implementation

In this section we specified in more details, how the diagnosis system works and was implemented.

The ProPAT_deBUG diagnosis system analysis the student program, after it has been compiled successfully. To derive the component/connection model from the program, we built a parser in ANTLR [14], a framework to construct recognizers, compilers, and translators from grammatical descriptions containing Java, C, C++, or Python actions. The diagnosis sub-tasks defined in the conceptual model (Section 2) were implemented as follow:

- **Symptom Detection of Programs.** Given a set of test-cases data, represented, for example, as a table of correct input/output values...
for the problem solution, a symptom is any difference between the program outputs and the outputs of the test-cases. If no symptom is detected, the diagnosis task will not be performed.

- **Hypotheses Generation/Find Contributors.** We implemented this task as a production rule system with a record of dependencies. The rules were constructed such that if a subset of input and output values are known, new values are computed and propagated. The simulation (constraint propagation) is done by disabling the behavior of the abnormal component (i.e., no values are propagated by it). This allows us to locate faults in expressions, such as wrong operators. Then, to determine the set of contributors in an inconsistency, a strategy of type ATMS was implemented: while the variable values are propagated through the production rules, we keep track of the components that had derived that value and which input values had been used (justifications). After all the values have been propagated, starting from the last contradiction, a backward search finds all constraints that helped to derive the contradiction [5] which will serve as the input for the Reiter’s algorithm.

- **Hypotheses Generation/Transform Contributors to Hypothesis Set.** The conflict sets are transformed into a hypothesis set using the Reiter’s Algorithm [15], i.e., constructing the hitting set tree, described in Section 2.3.

- **Hypotheses Discrimination.** The fault estimate value is derived from test case execution as outlined in [11]. To further guide diagnosis, before we start debugging, we first run all test cases and record which fragments are executed for each test. As every test case is either classified as correct or faulty, we obtain estimate values for each fragment, representing their likelihood to be faulty.

3.3 Hypotheses Discrimination through elementary patterns templates

After having the hypothesis set ordered by the estimate values, they can then be communicated to the student in that order by asking him about the expected values for the outputs of the components involved in the selected hypothesis. If the hypothesis bel lows to an identified elementary pattern, the system uses a template based on its documentation to establish a better dialogue with the student about his intentions. P.e., suppose that a hypothesis $H$ includes lines L3 and L6 that are part of the elementary pattern called counting repetition which has the following abstract solution structure:

\[
\text{L1 \, <INITIALIZATION>}
\]

\[
\text{L2 \, <COUNTER INITIALIZATION>}
\]

\[
\text{L3 \, while <COUNTER CONDITION>}
\]

\[
\text{L4 \, <READ GENERATE A SEQUENCE ELEMENT>}
\]

\[
\text{L5 \, <PROCESS ELEMENT>}
\]

\[
\text{L6 \, <UPDATE OF THE COUNTER>}
\]

\[
\text{L7}
\]

Therefore the dialogue with the student should be:

> We have detected that you used the elementary pattern Counting Repetition to construct your program through the ProPAt editor. The diagnosis system pointed out a possible error in lines 15 and 21 of your program which corresponds to lines L3 and L6 of the pattern. Those lines are responsible for the condition test and update of the counter variable in the pattern.

> For the test case T, the value of the counter variable after executing line L2 is 0. What is the value of the counter variable after the whole Counting Repetition pattern has been executed (i.e., line 32 in the student program)?

> If the student answer has the same value as the one calculated by the execution of the program for the test case T (we use the value based model to propagate those values), then the ProPAt_deBUG will be called again with this extra value observation. Otherwise, the student fault has been detected.

4 Empirical Results

In order to evaluate the diagnosis system, we have selected a set of programs with typical logical and semantic faults classified according to their consequences and its type. Dominik Wieland [17] proposes a taxonomy of programming faults that we have extended with a decomposition of the semantic faults in two new categories: semantic faults caused by the student wrong interpretation about the individual language constructions and semantic faults caused by the difficulties on the understanding of the interactions and coordination between multiple structures.

The goal of the evaluation of diagnosis system is to measure the ability of the system to return the hypotheses set that includes the real program faults without returning too many plausible hypotheses (i.e., hypotheses that can explain the observations but that are not the real faults). For our goals, since the precision and recall measures are usually defined for the information retrieval area, the sets used in these measures has been redefined for the evaluation diagnosis systems as follow:

\[
\text{FP} = \{\text{set of faults in the student program}\}
\]

\[
\text{HV} = \{\text{set of the k first hypotheses returned by the diagnosis system ordered by a hypotheses estimate value}\}
\]

**Precision (PRE)** : is the ratio of the number of correct hypotheses retrieved by the system ($|FP \cap HV|$) divided by the total number of hypotheses retrieved by the system.

\[
\text{PRE} = \frac{|FP \cap HV|}{|HV|}
\]

By definition, PRE is a measure between 0 and 1: $PRE = 1$, means that all the hypotheses given by the diagnosis system are real faults; $PRE = 0$, indicates that the diagnosis system were not able to find any real hypotheses.

**Recall(REC)** : is the number of correct hypotheses retrieved by the system divided by the number of faults of the student program.

\[
\text{REC} = \frac{|FP \cap HV|}{|FP|}
\]

That is, $0 \leq REC \leq 1$: $REC = 1$ means that all real faults were considered as hypotheses by the diagnosis system; $REC = 0$ indicates that the diagnosis system were did not considered any of the real faults as hypotheses. It is clear that what one expect from a diagnosis system is a balance between precision and recall.

**F-measure** : combines precision and recall and is defined by:

\[
F - \text{measure} = \frac{1}{\frac{\alpha}{PRE} + \frac{1 - \alpha}{REC}}
\]

Where $\alpha$, $0 < \alpha < 1$ is used as a weight factor to adjust the importance of the precision for the user. The better the F-measure is, the better the precision and recall are. For instance, if we use $\alpha = 0.5$, we give the same importance to the precision and recall, that is:

\[
F - \text{measure} = \frac{2 \times PRE \times REC}{PRE + REC}
\]
The ProPAT_DEBUG system was evaluated for 28 programs with different types of faults. The results indicate an essential aspect of the program diagnosis: it returns a small number of hypotheses about the student fault. The precision, recall and F-measure were calculated for the set HV with the k-first hypotheses ordered by the fault estimate value of the hypotheses for \( k \in \{1, 2, 3\} \).

Using \( k = 3 \), we observed that the F-measure has shown a better combination between precision and recall. Because these results are not very close to 1, we analyze the importance between precision and recall in the diagnosis, changing the value of \( \alpha \). The Fig. 3 shows the values of the F-measure for \( k = 3 \) and \( \alpha = 0.3 \), i.e., we give less importance to the precision than recall. The results are that the majority of problems (64%) has measure-F between 0.63 and 1. It is important to notice that the recall in the majority of tested programs is big for a little number of hypotheses (Figure 4).

The values of F-measure used in the evaluation of the system shown that the diagnosis is more difficult for structural logical (12 out of the 28 programs analyzed) faults in the student program, while the results were close to 1 for faults of type functional logical. That is, for this type of faults, the diagnosis system always returned the real fault as a hypotheses with few (1 or 2) more hypotheses different from the real.

The purpose of diagnosis is often said to explain or indicate the cause of errors made by the learner. Like in [6] we focus on "detecting bugs rather than misconceptions".

5 ProPAT and Intelligent Tutoring Systems

ProPAT was first created to be a special programming environment based on elementary patterns and not an intelligent tutoring system. Nevertheless, with the addition of the model-based diagnosis system, the ProPAT has gain some characteristics of intelligent tutors discussed in the next subsections.

5.1 Model Based Diagnosis and learner behavior

The research presented in this paper is concerned with the local part of learner modeling, that can be divide into three major functions: monitoring (performance assessment), diagnosis (behavior diagnosis) and repair (explanation generation) [6]. In ProPAT the performance assessment is made in two different ways: (1) by monitoring the student insertion of elementary programming patterns into the editor and (2) by running the bench-mark examples looking for symptoms (incorrect outputs). The diagnosis (behavior diagnosis) is made by the ProPAT_DEBUG and the repair (explanation generation) can be made by using the programming patterns documentation.

The purpose of diagnosis is often said to explain or indicate the cause of errors made by the learner. Like in [6] we focus on "detecting bugs rather than misconceptions".

5.2 ProPAT and PROUST

ProPAT, seen as an ITS, can be compared with PROUST [10] [9]: an ITS for programming learning from the 80s which development was based on the psychology programming theory. PROUST represents an important landmark in the evolution of the programming tutors. PROUST tries to identify the intentions of the student through the matching of the problem sub-goals and plans in the student program.

An example of problem solved by PROUST [10] is:

**Problem 1:** Read numbers, taking their sum, until the number 99999 is read. Report the average. Do not include the final number 99999 in the average.

To solve this problem, PROUST recursively tries to decompose the problem goals into sub-goals and by consulting an internal library of plans (also composed by subgoals and parts of code) tries to match plans to solve the problem goals and subgoals. The Figure 5 adapted from [10] shows how that decomposition is related with the correct solution code developed in C.

Although the PROUST has been used as the inspiration for many others programming tutoring systems, it has serious limitations that did not let it to be effectively used in programming learning. Some of them are:

- PROUST requires that an experienced programming teacher specifies a complete plan library, which corresponds to all possible ways a student can solve a problem.
- In the original proposal it is not possible to add new plans in the library, i.e., the library is previously designed by consulting expert educators.
the student does not have access to the library of plans so the system communication interface is the only responsible to promote the student learning.

In the PROPAT system the subgoals problem are not represented and the plan library is replaced by an elementary programming patterns library, that can be accessed directly by both: the student and the teacher. The library is used as a learning material and increasing the probability of the student using the pattern in his program. The patterns can be defined by the teacher, and different to PROUST, they do not need to correspond to all correct forms to solve problems, since we are able to monitor (performance assessment) and observe which elementary pattern the student has inserted in his program.

6 Conclusions and Future Works

We have presented a programming environment, called PROPAT that allows the student to program using elementary patterns, and uses a model based diagnosis system to detect a fault in a student program, named PROPAT-deBUG.

The main goal of this work is to present some empirical experiments elaborated to evaluate the diagnosis system. That is: to measure the ability of the system to return the hypotheses set that includes the real program faults without returning too many plausible hypotheses, i.e., hypotheses that can explain the observations but that are not the real faults. Too many hypotheses would confuse the student during the discrimination process, while few of them without including the real fault would not make the student learn from his own mistakes. Further, each hypothesis is communicated to the student through elementary patterns templates.

The diagnosis system was evaluated against 28 programs with different types of faults according to a classification of typical student faults. The values of F-measure used in the evaluation of the system showed that the diagnosis is more less efficient for structural logical faults in the student program. However, the results were close to 1 for faults of type functional logical. That is, for this type of faults, the diagnosis system always returned the real fault as a hypotheses with few (1 or 2) more hypotheses different from the real. The results show that during the interactive debugging process it is possible for a student to learn by answering the questions posed by the AI diagnosis system to discriminate its fault hypotheses about the program.

With this work we conclude that, for detecting student’s functional logical faults, it is possible to have an automatic debugger that returns a small set of possible faults and that always includes the real fault. However, we know that the most fruitful task for the near future is to run a bigger number of tests and analyze how a group of students, can improve learning using this tool.

Another task for the near future is to work more on the elementary patterns’ based dialogues. The idea is to extend the number of elementary patterns in PROPAT and to construct natural language templates to communicate the fault hypotheses based on the patterns documentation for them.

REFERENCES