Programming Learning: A Hierarchical Model Based Diagnosis Approach

Wellington R. Pinheiro¹, Leliane N. de Barros¹, Karina V. Delgado¹

Institute of Mathematics and Statistics – University of São Paulo

Key words: ITS, Pedagogical Patterns, Model Based Diagnosis, Program Debugging.

Abstract

Model based diagnosis systems from Artificial Intelligence are used to find faulty components in physical devices and have also been applied to find bugs in computer programs, also called "model based software debugging". Although this approach can help an advanced programmer to find bugs in his program, it seems not adequate for programming apprentices. In this paper, we present an approach for program debugging that explores the idea of abstract components to perform a hierarchical model based diagnosis. Abstract components of the student's program can be a function, a procedure or a programming elementary pattern. The program bugs hypotheses generated for each abstraction level are then communicated to the student, improving this way his learning opportunity.

1 Introduction

Learning to program computers is hard: it involves to learn a programming language and how to solve problems by writing program solutions in that language. Even though a student already knows the syntax and semantics of individual sentences of the language, research results show [19] that it is still very difficult for the student to learn how to use the sentences to construct a program solution, i.e., to specify his problem solving method in the programming language.

An Intelligent Tutoring System (ITS) [17] is an e-learning tool that uses artificial intelligence techniques, such as, knowledge representation techniques and problem solving methods, e.g., diagnosis or planning methods. One important aspect of an ITS is the representation of the student knowledge, also known as the student model or student diagnosis [16], which should be constructed based on student's problem solutions, i.e., based on the diagnosis of student's solutions. For example, in the area of programming learning, the student's problem solution is a computer program. It is based on the student model that an ITS should be able to make its instructional decisions (e.g., when to change to a new topic or which is the next topic to be learned). A well known ITS for programming, called PROUST [10], constructs the student model by trying to recognize, in the student's computer program, his programming plans and goals (intentions): given a student's program solution, possibly with a failure, the system tries to match parts of it with programming plans retrieved from a library of programming knowledge. A drawback of this approach is that stored plans had to be provided by the system designer with no guarantee to cover the whole set of possible solutions for a problem.

¹ This work was in part supported by the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) under project 2005/60647-1 and by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES).
In previous work, we have proposed a system [3, 8] that uses model based diagnosis [6, 15] capable of finding some types of faults in the student's program. The main goal of this system is to allow the construction of the student model based on the student's program diagnosis, without using a library of programming plans and goals, previously specified for a given set of programming problems.

Model based diagnosis (MBD) is the task of finding faulty components in a specific device, typically applied to troubleshooting physical devices. Model Based Diagnosis involves three sub-tasks: symptom detection, hypothesis generation and hypothesis discrimination [4]. The diagnosis is actually done through the hypothesis generation sub-task, as follows. On one hand, we have the actual device (composed by a set of components - C) whose behavior can be observed, probably with a failure; on the other hand we have the correct device model (system description - SD) which is used to make predictions about the device behavior. Such model, typically describes the components of the system, their connections and their individual behavior. The difference between an observation (OBS) and a prediction is called a discrepancy; while a match is called a corroboration. Both discrepancies and corroborations are used to identify which parts of the device are possibly faulty [4].

MBD has also been applied to program debugging [11, 18, 12]. 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 “tutor” tries to understand the differences between the student’s program code and his intentions (possibly identified by interacting with him). If we see the student’s program as the system (SD) we want to perform the diagnosis, it is interesting to notice that we do not have the correct program model to make predictions, but the faulty student’s program. The idea is to use the student extra information to detect the wrong parts of his program, i.e., according with his intentions.

Model based diagnosis for program debugging, also called model based software debugging, has been originally proposed by Franz Wotawa and Markus Stumptner [11] to help advanced programmers to find bugs and not to support programming learning. In the program model, language expressions and sentences are represented as components; while the connections between components represent information flow. Components behavior are described based on the language semantics. The diagnostic is a set of possible faulty components, in the case of programs, the diagnostic is a set of hypothesis on the expressions or sentences of the program code with bugs.

An interesting aspect of the student’s program debugging, besides given information to construct a student model, is that it can, on its own, promote learning. This is obviously a side effect gained in all learning methodologies centered on problem solving. In this work we will focus on this aspect, instead of the student model construction, necessary by an ITS to make its instructional decisions. That is, we will explore possible ways to interact with the student during a model based diagnosis in order to promote programming learning.

The difficulties and drawbacks of applying model based software debugging [11] to support programming learning are:

1. for a regular size program, we usually have too many hypotheses of failure to be further discriminated [4]. The discrimination process involves asking the student for extra information about his program (i.e., to acquire the student's intentions, e.g., his expected values for variables during the program execution). Too many hypotheses to be discriminated would make the student tired or confused;
2. pointing out the program lines, as the faulty components hypotheses about the student's program (as in [11]) gives little or no information to the student to find out its mistakes or misconceptions. Our claim is that this is definitely not the best way to interact with the student; and
3. novice students don't know how to express his programming intentions and do not have enough programming knowledge in order to supply detailed information about the behavior of his own program, unless he could refer to his intentions and general strategies in a suitable language.

In this work we propose a new method for software debugging which should be more appropriate to be used in a programming learning tool. Therefore, we extend our previous work on MBD of programs [3, 8] with a hierarchical approach for program debugging [13, 5]. We claim that, by introducing the notion of abstractions and hierarchical refinements, we can provide better results on the three limitations discussed above.

This paper is organized as follows. In Section 2 we present background concepts about model based diagnosis, model based software debugging and hierarchical model based diagnosis (HMBD). Section 3 presents our proposal of defining program abstractions and apply HMBD in order to promote programming learning. Finally, in Section 4 we present our conclusions.

2 A brief review

2.1 Model based diagnosis: the general theory

Diagnosis is the task of identifying the cause (an explanation) of a fault that manifests itself through some observed behavior [4]. The whole process involves three main sub-tasks: symptom detection, hypothesis generation and hypothesis discrimination. In particular, the hypothesis generation can be formally described as follows [15].

Definition 1.1 A system is defined as 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 1.2 Given an observation, \(OBS\), the triple \((SD,COMP,OBS)\) is a diagnosis problem for the system \((SD,COMP)\), with observation \(OBS\).

Definition 1.3 A diagnosis \(\Delta\) for the system \((SD,COMP)\) is a minimal subset of \(COMP\) such that:

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

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

Definition 1.4 A contributors set for \((SD,COMP,OBS)\) is a set \(CO \subseteq COMP\) such that: \(SD \cup OBS \cup \{\neg AB(C) \mid C \in CO\}\) is inconsistent.

† In logic, a consistent formula is a formula that holds in at least one interpretation of the world.
Definition 1.5 A hitting set for a collection of sets $F$ is a set $H \subseteq \bigcup_{S \in F} S$ such that $\forall S \in F, H \cap S \neq \emptyset$, i.e., a hitting set is a set that intercepts all the sets of the collection $F$. A minimal hitting set is a hitting set such that none of its subsets is also a hitting set. When the collection set $F$ corresponds to the set of all contributors sets $CO$ for a diagnosis problem $(SD, COMP, OBS)$, the minimal hitting set is the simplest possible explanation for the observations. The next theorem shows a constructive form to find the hypotheses from the contributor collection set.

Theorem 1.1 The set $\Delta \subseteq COMP$ is a diagnosis for $(SD,COMP,OBS)$ if and only if $\Delta$ is a minimal hitting set for the collection of all contributor sets for the diagnosis problem $(SD,COMP,OBS)$ [15].

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 $\emptyset = \emptyset \cap H(n)$. 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.

2.2 Model based software debugging

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++ 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. In this work we are focusing on finding functional program bugs and special cases of structural bugs.

The basic idea for diagnosing programs, instead of diagnosing physical devices, is to derive a system description ($SD$) directly from the student's program and the programming language semantics. This model must represent components, connections and its behavior, based on the actual student's program behavior which reflects its errors. The observations ($OBS$) are the incorrect outputs in different points of the original program code. There are two approaches that can be used for program modeling: a value-based model [11] and a dependency based model [18].

The Value-based Model

In the value-based model, expressions and statements are represented as components and the semantics of the expressions and statements are described by sets of logical sentences
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, for example, 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 [11]. The logical sentences defining these connections correspond to the student model of the program. 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 a assignment's left side;
- sentences, assignments, conditionals, while loops, return statements and expressions, are mapped into components.

![Figure 1](image_url)

**Figure 1** Simplified structural and behavioral models of the program sentence: \( \text{start} = \text{from} \times x \). The sentence is modeled by two components: the IntegerMultiplication (C0) and the Assignment (C1).

The example of structural model on Figure 1(a) shows how to construct the value-based model for the sentence \( \text{start} = \text{from} \times x \). An assignment component has two ports: input and output port. The input port is related to the evaluation of the expression on the right side of the assignment statement and the output port is related to the variable on the left side of the assignment. The multiplication operator is mapped to a component with two inputs and one output port; the output port is related to the evaluation of the relation between the two operands induced by the operator. In order to obtain a complete system description that can be used by the diagnosis algorithm, the behavior of all individual fragments of the computer language must be specified. Since each fragment corresponds to a single construction of the programming language, the behavioral model can be derived from the language specification. Figure 1(b) shows the behavioral model for the Integer-Multiplication component \( C0 \) and the Assignment component \( C1 \), where \( \text{in}_1(C0) \) and \( \text{in}_2(C0) \) represent the two inputs of the component \( C0 \) and \( \text{out}(C0) \) is the output of the component \( C0 \).

**Program repair versus student misconception repair**

Although using MBD for software debugging has been proved to help an advanced programmer to detect faults in his program, a novice student will probably not be able to recognized his faults or learn from that. Research also shows that by identifying the reasoning
steps used by a novice student, an ITS would be able to communicate with the student in a better way, and can help him to understand what he has done wrong [7]. This requires interaction between the diagnosis process and the student in terms of his reasoning steps, and not only by pointing out the possibly faulty program sentences.

3 Hierarchical Model Based Diagnosis for Program Debugging

One of the main motivations of using hierarchies on model based diagnosis is to achieve performance gains when applied to physical devices. However, for program debugging, we can also explore hierarchies to communicate the diagnosis in different levels of abstraction to the student. In order to do so, we need to recognize abstractions (i.e., abstract components) in computer programs, which can be:

- procedures or functions;
- *elementary patterns*, that are strategies to solve recurring programming problems which can help novice students to learn how to program. Elementary patterns are recommended by programming educators for novice students [14].

The use of abstract components for a programming ITS can bring some advantages, such as:

- to establish a dialog with the student in terms of problem solving strategies, in a high level communication language through the elementary patterns, functions or procedures;
- to allow reasoning about a program in a hierarchical fashion, i.e., to detect program faults in different levels of abstraction.

Figure 2 presents a student program in Java language that solves the *even-odd-sum* problem: *Given a set v of integer numbers, calculate the difference between the sum of the even numbers and the sum of the odd numbers.* It is possible to recognize in this program the elementary pattern known as *counted loop* [1] composed by the line 3 and the lines from 6 to 13. This elementary pattern responsible for iterate over a collection of *n* elements and process each one of them. Figure 3(a) illustrates the value-based model of the student program from Figure 2, called *base model*. Figure 3(b) contains an abstract component representing the counted loop elementary pattern.

We claim that the model based diagnosis over such abstract component will overcome some of the limitations of applying traditional model based debugging in a programming learning tool. In the Section 3.1 we give an introduction to hierarchical model based diagnosis of physical systems and in Section 3.2 we present our proposal of a new algorithm of hierarchical model based diagnosis for programming learning, that will be used in a tool called *H-Propat*.

3.1 Hierarchical Model Based Diagnosis

Abstractions in MBD have been claimed to be a way to reduce computational cost of finding a diagnosis [9, 13]. The use of abstractions makes possible to describe a system in a simplified view, while preserving some important characteristics. The way that abstractions are commonly used in a system description is by describing hierarchies.

*Hierarchical model based diagnosis* (HMBD) is an approach of MBD that uses multiple levels of detail to describe a system. Each level, except the first, is composed by a set of abstractions constructed upon a more detailed level. Basically, two kinds of abstractions are considered [5]: (i) *behavioral abstractions*, that are defined by grouping behavioral descriptions of the
system, e.g., digital circuits work with values 0 and 1, but internally some components may work with analog values; (ii) structural abstractions, that are constructed by aggregating a set of components, e.g., a digital circuit is composed by transistors, resistors, capacitors, etc. A component defined by a structural abstraction is known as an abstract component. We call base model a model without any abstract component.

```
public int evenMinusOdd(int v[])
{
    int n = v.length;
    int counter = 0;
    int evenSum = 0;
    int oddSum = 0;

    while (counter < n) {
        if (v[counter] % 2 == 0) {
            evenSum = evenSum + v[counter];
        } else {
            oddSum = oddSum + v[counter];
        }
        counter = counter + 1;
    }

    return evenSum - oddSum;
}
```

Figure 2 A student's program in Java language that solves the even-odd-sum problem.

An abstract component replaces a set of components by only one component and its specifications is comprised of:

- an internal structural model, describing how its aggregated components are connected. This model is composed only by the sentences describing the connections that links two internal components;
- a behavioral model, describing how the abstract component must behave in terms of its aggregated components, i.e., it should be modeled in such way that it represents the global behavior defined by its aggregated components;
- an external structural model, that describes how the abstract component is connected with the outside components, i.e, the components that do not belong to the abstract component.

Figure 4 presents two models for a digital circuit (which operates only with values 0 and 1). The model on Figure 4(a) is composed by an AND component, $A_1$, that executes a boolean and operation; and an inverter, $I_1$, that outputs the opposite value of its input, i.e., 0 if the input is 1, or vice-versa. Figure 4(b) presents the abstract component $NA_1$, that is composed by the aggregation of components $A_1$ and $I_1$.

Suppose that $C$ is an abstract component composed by the aggregation of the components \{\(C_1, C_2, \ldots, C_k\)\}. The behavior of $C$ must be consistent with the following axioms [2]:

\[
\neg AB(C) \rightarrow \neg AB(C_1) \land \neg AB(C_2) \land \ldots \land \neg AB(C_k) \tag{Axiom 3.1}
\]

\[
AB(C) \rightarrow AB(C_1) \lor AB(C_2) \lor \ldots \lor AB(C_k) \tag{Axiom 3.2}
\]

Axiom 3.1 states that if the abstract component is not faulty, then all the aggregated components must not be faulty. This axiom can prevent some faults to be detected in abstract components, in the case that an internal faulty component counterbalances the influence of
another internal faulty component. The Axiom 3.2 states that if the abstract component is faulty, then at least one of its aggregated component must be faulty.

![Diagram](image)

**Figure 3** The base and abstract models for a program that solves the even-odd-sum problem. The abstract model contains an abstract component based on the elementary pattern called *counted loop* [1].

Given a hierarchical multi-level description of a system, the method used in HMBD to find the hypotheses consists on mapping each level of detail into a diagnosis problem and to use a traditional MBD (Section 2.1) [5] to find the diagnosis hypotheses for each level at time. Starting from the more abstract level, the hypotheses found at a certain level are used to exclude components from the more detailed levels (Axiom 3.1). Thus, when the most detailed level is being diagnosed, a certain number of components will not be part of the search space.

![Diagram](image)

**Figure 4** The digital circuit on part (a) is composed by an *AND* component, $A1$, and an inversor component, $I1$. The part (b) shows the abstract component $NA1$, constructed by the aggregation of the components $A1$ and $I1$.

### 3.2 Hierarchical approach to detect faulty sentences in the student’s programs

In the traditional approach of model based software debugging (Section 2.2) the interaction with the user occurs only during the hypothesis discrimination. When considering hierarchical model based diagnosis for program debugging, the interaction can also occurs in each level of abstraction. Thus, hypotheses at each level of abstraction are shown to the student that can:

1. try to recognize his misconceptions, coming up with a new solution by rewriting his program. In this case, if the modified program is correct the debugging process is
finished. Otherwise, he can either resubmit his modified program to a new debugging session or to go back to the previous version of his program and select another hypothesis;

2. ask for help to discriminate the hypothesis set at the current level by supplying additional information (i.e., extra observations) about his intentions. In this case, the system would present, one by one, the most probably hypothesis to the student;

3. refine an abstract component in order to perform diagnosis in a more detailed model. In this case, a new hypothesis set is generated considering the components in a lower level.

Algorithm 1 specifies the interactive approach. This algorithm uses the functions: ConstructHierarchicalDescription, responsible for deriving a hierarchical model from a given program and its test suite; and Propat_MBD, that is a traditional model based diagnosis system (Section 2.1) we have already implemented [3, 8].

```
HMBD-Program-Debugging(P,T)
input: P: the student's program; T: the test suite for the program P.
1  M ← ConstructHierarchicalDescription(P, T)
2  CurrentModel ← the most abstract level of M
3  OBS ← T
4  Hyp ← Propat_MBD(CurrentModel, T)
while Hyp is not empty do
  5  show Hyp to the student
  6  if the student has modified the program then
      7     if the modified program corrects all faults then finish the debugging process
      8     else if the student wants to go back to the previous version of his program and try with the same hypothesis set then
      9         go back to the previous version of his program
     10  else  ▶ the whole process restarts with his new program.
     11     M ← ConstructHierarchicalDescription(NewProgram, OBS)
     12    CurrentModel ← the most abstract level of M
     13    Hyp ← Propat_MBD(CurrentModel, OBS)
  14  else if the student has decided to refine an abstraction Abs from the current hypothesis set then
      15     CurrentModel ← CurrentModel with Abs refined
      16     Hyp ← Propat_MBD(CurrentModel, OBS)
  17  else ▶ collect information from the student to discriminate the current hypothesis set
  18      OBS ← OBS union with the new observations from the student
  19      discriminate the current hypothesis set considering OBS
  20  end

Algorithm 1 A hierarchical approach to debug programs considering the student interaction defined in the three presented steps.

The main difficulty we have to face, given we already have the MBD system, is to construct the set of abstract program components for a selected instructional context (e.g., high-school class).

4 Conclusion

In this paper we present a method for program debugging that can overcome some of the limitations of the classical model based debugging when applied to programming learning. Our method considers the use of hierarchical model based diagnosis and the use of abstractions in the student’s program that are: elementary patterns, functions and procedures.
The abstractions can be used to establish a dialog with the student in terms of problem solving strategies, in a high level communication language.

The new algorithm of hierarchical model based diagnosis for programming learning proposed in this paper, considers the interaction with the students during the program debugging discriminating hypotheses in different abstraction levels.

The proposed algorithm describes a new way to refine abstract components, which is different form the algorithms proposed for physical devices [5, 13]. This algorithm will be implemented in a tool named H-Propat, which will use the features already implemented in Propat [3, 8].

References


Authors

<table>
<thead>
<tr>
<th>Wellington, Pinheiro, Bsc.</th>
<th>Leliane, Barros, PhD</th>
<th>Karina, Delgado, Msc.</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:wrp@ime.usp.br">wrp@ime.usp.br</a></td>
<td><a href="mailto:leliane@ime.usp.br">leliane@ime.usp.br</a></td>
<td><a href="mailto:kvd@ime.usp.br">kvd@ime.usp.br</a></td>
</tr>
</tbody>
</table>

Institute of Mathematics and Statistics – University of São Paulo,
Department of Computer Science

Rua do Matão, 1010 - Cidade Universitária - CEP 05508-090 - São Paulo - SP - Brasil