University of Bamberg Press

Bayerkuhnlein, Moritz; Wolter, Diedrich

Diagnosing Algorithms by Abduction from Manual Simulation

In:

Schmid, Ute; Leidner, Jochen L.; Kohlhase, Michael; Wolter, Diedrich (Eds.), Proceedings of the Second Work shop on Artificial Intelligence for Artificial Intelligence Education (AI4AI Learning 2024), Bamberg: University of Bamberg Press, p. 49-58. 2025. DOI: 10.20378/irb-107661

Bookpart - Published Version

DOI of the Article: 10.20378/irb-108887

Date of Publication: 07.07.2025

Legal Notice:
This work is protected by copyright and/or the indication of a licence. You are free to use this work in any way permitted by the copyright and/or the licence that applies to your usage. For other uses, you must obtain permission from the rights-holder(s).

This document is made available under the **Creative Commons Lizense CC BY.**

This licencse information is available online:
https://creativecommons.org/licenses/by/4.0/

Diagnosing Algorithms by Abduction from Manual Simulation

Moritz Bayerkuhnlein and Diedrich Wolter

Institute for Software Engineering and
Programming Languages
University of Lübeck, Germany
{firstname.lastname}@isp.uni-luebeck.de

Abstract

Taking a conceptual idea and turning it into a precise algorithm is at the heart of computational thinking. However, novice programmers often struggle when their code does not behave as *they* intended. This paper identifies problems in pseudocode algorithms based on deviations from observations that could be gathered during manual simulations by the programmer. By applying model-based diagnosis to the faulty pseudocode, informed by manual simulation, we locate errors and suggest fixes. The diagnosis results in the identification of a specific location in the algorithm and provides an output description for the faulty part that matches the programmer's intent during the manual simulation, thus aiding in debugging.

Introduction

Creating an algorithm starts with a rough idea that leads to a detailed executable implementation. Inexperience, misconceptions, or even a slight lapse in concentration can cause a programmer to get the details wrong, leading to a faulty implementation. Especially for novices, this can be a barrier to entry into programming in general.

A human programming tutor is adept at understanding both the written program and the student's intentions. Program comprehension involves

developing a mental representation of the program, to identify any errors and determine how to fix them. When pointing out a particular error in the source code, the tutor will usually provide an *explanation* of how a particular line led to an unintended action, and what the correct action should have been to achieve the expected result.

Techniques from Artificial Intelligence (AI) (Shapiro, 1982; Wotawa et al., 2002) and Automated Debugging (Dovier et al., 2005; Weiser, 1984; Zeller and Hildebrandt, 2002) can provide assistance and help to *diagnose* suspicious lines in the source code. However, these techniques are aimed at experienced programmers who can take measures to fix the code once the fault is located.

Novices, particularly in a practice setting, may need more help to determine why a certain statement caused problems, especially when they have an inadequate *mental model* (Johnson-Laird, 1989) of the language or the data structure used and regard the algorithm produced as correct (Chmiel and Loui, 2004).

Here, we use model-based diagnosis techniques to diagnose faulty implementations based on the programmer's intent to master a given programming exercise. To do this, we trace a manual simulation of dominant data structures to capture the intermediate steps the programmer intended their program to go through. With this approach, we aim for a mental model aware fault diagnosis, where a source-code location that is identified as potentially faulty gets justified and aligned with the expected program behavior.

Preliminaries

When debugging software, we find discrepancies by testing the potentially faulty program Π with the correct input and output specifications (I, O). Fault localization techniques that rely on source code execution, such as Spectrum-Based Fault Localization (SFL) (Abreu et al., 2009), highlight code locations based on their co-occurrence with unexpected

outputs. While these methods are efficient, they cannot isolate the underlying reasons why a statement may have produced an unexpected result.

The *inference to the best explanation* of an unexpected outcome, so-called abduction can reveal possible reasons using AI techniques (Magnani, 2011). In *Model-Based Diagnosis* (MBD) (Reiter, 1987), a system model, made form a set of connected components, predicts behavior based on inputs and is restricted by observed outputs. When a failure occurs, the model identifies discrepancies between the expected and actual output. A diagnosis is the set of components that, if assumed faulty, would *explain* the failure.

Formally, our goal is to identify the component of the source code $COMP$ in the form of a statement or a group of statements that could have caused the mismatch by labeling it *abnormal* (ab) (Console et al., 1993; Wotawa et al., 2002).

More formally this is specified by Reiter's Consistency-Based Diagnosis (Reiter, 1987), here using notation from Model-Based Software Debugging (Wotawa and Dumitru, 2022) using constraint solvers, where $M(\Pi)$ and $\mathrm{M}(\mathrm{I},\mathrm{O})$ is the translation of a program Π and test case (I,O) into a logical model and reasoned about using a constraint solver.

Definition (Consistency-Based Diagnosis Problem): Given a program model $\langle M(\Pi), M(I,O), COMP\rangle$, compute all minimal sets $\Delta \subseteq COMP$ of abnormal components such the following expression is satisfied:

$$M(\Pi) \cup M(I,O) \cup \{ab(c) \mid c \in \Delta\} \cup \{\neg ab(c) \mid c \in COMP\backslash\Delta\}$$

Imperative programming languages and algorithms use variables that collectively represent the *state* of a program and establish dependencies (Weiser, 1984). Datatypes restrict the possible values of these states to a (finite) domain. Therefore, any diagnosis Δ must also find a satisfactory assignment for the affected variables; otherwise there would be no fix for a single statement (Bayerkuhnlein and Wolter, 2024).

Representing Algorithms for Diagnosis

In previous work, we modeled system descriptions using test cases that target individual functions as program components (Bayerkuhnlein and Wolter, 2023). For algorithmic diagnosis, the model structure in $M(\Pi)$ is based on the program's control flow.

Given an imperative program Π we parse its operations and control structures to create a complete flow graph $G(\Pi) = \langle COMP, E \rangle$, where nodes $COMP$ correspond to diagnosable statements. As control structures we consider *if* statements for selection and *while* loops for iteration; function and recursive calls are left for future work. Directed edges E represent the control flow of Π. The figure below illustrates this representation.

Each flow line in E marks a *state transition* during the execution of Π. In terms of a system description, this is how statements (components) are connected. Representing these transitions as relations allows inversion and reasoning about alternatives (Ross, 1997).

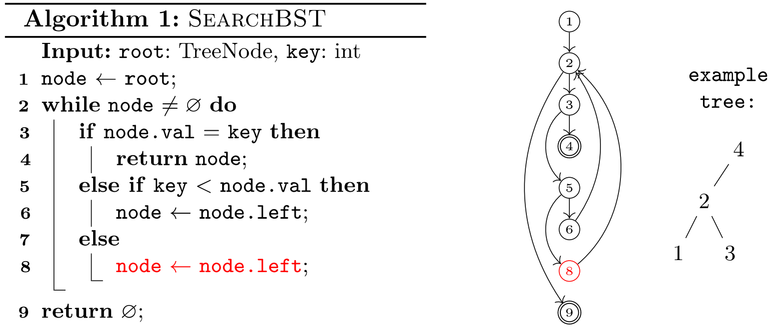


Figure 1: Algorithm with error and corresponding Control Flow-Graph

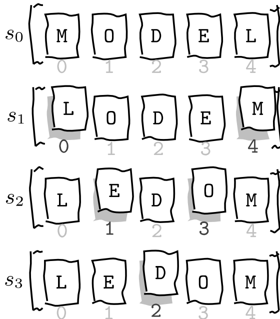
All variables used within Π are represented by their value $Var(\Pi)$ in a structure that represents the current state of the program, e.g. the shown algorithm *SearchBST* in Figure above has states of form $s = \langle root, key, node \rangle$.

Each node in G represents a statement in Π and is a component of $COMP$, making it a potential part of a diagnosis. Every statement defines behavior for input and output states, affecting the program pointer pc and potentially altering variable values $Var(\Pi)$. Transitions reflect control-flow manipulations, such as *if* statements. If a manipulation constrains state transitions to the point of impossibility, a conflict arises. This conflict is resolved by marking the statement as *abnormal*, thus identifying a diagnosis. In the above example, any right subtree cannot be found since line 8 updates the *node* variable incorrectly. To account for an observation (I, O) modeling *SearchBST(example_tree, 3) = node 3* instead of ∅, a fix of line 8 must transition node to node 3.

This approach presents challenges, as *any* state transition might be attributed to abnormal components, making it difficult to explain their behavior logically. The issue is worsened when loops are involved, as affected statements run multiple times. Inducing potential values along a sequence, such as program execution, faces under-constrained issues; abnormality at the very end of a program could explain any test result, regardless of prior actions. In practice, model-based software debugging is typically used to identify minor issues resulting from an experienced programmer's oversight (Wotawa and Dumitru, 2022).

Diagnosing from Manual Simulation

There are several conceptual solutions to a programming problem, each with various implementations. A test case cannot fully capture the programmer's intent, especially for algorithms with a single output per input, making internal workings opaque. Correct implementations appear identical in test cases, though they may differ in implementation. Faulty implementations can hide problems until later in the execution.



Algorithm 2: ReverseString

```
  Input: word: String
1 left ← 0;
2 right ← len(word) - 1;
3 while left < right do
4 |   word[left] ← word[right];
5 |   word[right] ← word[left];
6 |   left ← left +1;
7 |   right ← right −1;
8 end
```

Figure 2: String Reversal Manual Simulation and faulty Algorithm

Running Example (Reversing Strings I). *The student is tasked with implementing a procedure to reverse strings. Possible strategies include: (i) Copy the string to another array starting from the end, (ii) Swap elements in pairs, working towards the middle, or (iii) Use a stack to push and pop elements back into the array. Choosing strategy (ii), a student might find that manual simulation with the input string "MODEL" correctly results in "LEDOM". However, the implementation unexpectedly produces "LEDEL", indicating a discrepancy from the manual simulation. To suggest a correction, a tutor could pinpoint lines 4 and 5, noting that the implementation failed to execute the expected swapping behavior due to prematurely overwriting the left side.*

Solving tasks such as the one described is inherently a creative process (Knobelsdorf and Romeike, 2008). It involves first devising a general solution to the computation to be performed and then expressing that solution in a programming or pseudo-code language. People also enjoy developing *their own* solutions (Sharmin, 2021).

For absolute novices, an Intelligent Tutoring System (ITS) with model tracing, which uses a cognitive model to follow a student's problem-solving steps, provides helpful feedback by tracking and offering corrections (Anderson, 1993). However, modeling the vast domain of programming is challenging and can limit intermediate students by enforcing overly prescriptive solutions (Johnson and Soloway, 1985).

The aim of this approach is to respect the creative integrity of the student and to allow for multiple or varied strategies while still giving them some guidance to achieve their goal. Assisting novice programmers goes beyond finding and fixing bugs, but to identify and resolve the exact problem that caused the discrepancy between the conceptually intended program, their mental model, and the realized program.

Formally, we define the intention of a programmer as $\langle \hat{\Pi}, \hat{\Sigma} \rangle$, where $\hat{\Pi}$ represents the mental model of the program and $\hat{\Sigma}$ represents the conceptual semantics, i.e., how the programmer thinks the components $c \in COMP$ operate. In contrast, the implemented program is denoted as $\langle \Pi, \Sigma \rangle$, where Π is the executable program and Σ represents the semantics of the programming language. Behavior of an individual component is denoted $\Sigma(c) \in \Sigma$.

Ideally, both $\hat{\Pi}$ and Π should exhibit identical behavior, as $\hat{\Pi}$ is the conceptualization of Π. However, discrepancies in behavior can arise due to implementation errors or misunderstandings of the program semantics (Chandra et al., 2024). Our objective, upon identifying such discrepancies, is to isolate components Δ within Π that deviate from the intended behavior $\langle \hat{\Pi}, \hat{\Sigma} \rangle$. We aim to propose suitable alternatives $\hat{\Sigma}_\Delta = \{ \hat{\Sigma}_\Delta(\mathrm{c}) \mid \mathrm{c} \in \mathrm{COMP} \}$ to align the program with its intended behavior, facilitating program repair or further analysis. The feasibility of synthesizing a program repair Π' that implements the proposed behavior $\hat{\Sigma}_\Delta(\mathrm{c})$ using Σ is not addressed here.

A conceptual program $\hat{\Pi}$ is low fidelity, representing only the conceptually relevant dominant data structures D during manual simulation. Observations from the manual simulation are represented as partial state descriptions, focusing solely on the most relevant data manipulated during algorithm execution. The remaining variables relevant to $M(\Pi)$ act as scaffolding, bridging the gap between conception and implementation.

We define the problem of inducing the intended program state from observations as follows.

Definition 2 (Intended Program Abduction). Given an abnormal component $c \in \Delta$ a model of a program $M(\Pi)$ and observations $M(I, O)$. We search for an alternative semantics $\hat{\Sigma}$ which transition the state of the program s_i into s_{i+1} following the operation of c, such that $M(\Pi) \cup \hat{\Sigma}_\Delta(\mathrm{c}) \vdash M(I, O)$.

Applying the diagnosis to a single test case of the running example, focusing only on input and expected output, we gather a diagnosis $\Delta = \{5\}$. However, some of the justifications are not intuitive. For example, a naive solution with $\hat{\Sigma}(5)$ would replace the entire string in the last loop iteration, with the final output. While this solution is consistent, it is clearly not what the programmer intended.

The conceptual program $\hat{\Pi}$ is intangible, and a programmer's verbal descrip- tion of their mental model can be imprecise with implicit assumptions. Instead, we propose manual simulation as an interface between the program's conception and implementation, integrating it into the diagnostic model through the data D used. The conceptual program $\hat{\Pi}$ can execute as a mental (Forbus, 1990) or manual simulation, producing a trace of states that act as observations for the intermediate states of the actual implementation.

Running Example (Reversing Strings II). Consider a situation where a pro- grammar sketches out a trace of the program on a whiteboard by drawing the array (dominant data structure) and simulates its states s0 ... s3 step by step (see Fig. 2). On the whiteboard, the change in indexes for the two swapping positions are omitted, and managed they act as scaffolding for the implementation.

With intermediate observations from the manual simulation, the diagnosis $\Delta = \{5\}$ remains consistent, but the abduced semantics $\hat{\Sigma}(5)$ can only justify assignments by incorporating intermediate states s1 ... s3 (see Fig. 2). The remaining variables relevant to $M(\Pi)$, acting as scaffolding, are then subject to abduction and reconstruction by constraint programming.

This process, through abduced values, outlines what a potential program repair might be.

Conclusion

Manual simulation on paper, whiteboards or through conversation is essential for understanding or developing algorithms. It also generates intermediate observations that reflect the programmer's intention of the algorithm's behavior, making it an ideal source of information to diagnose programming mistakes.

We propose to use manual simulations, treating them as observations of computational states to inform diagnostic methods for debugging algorithm construction and implementation. These intermediate observations further constrain consistency-based diagnostic methods and help to derive potential fixes for problematic areas. In addition to a prototype implementation for diagnosing simple data structures, future work will include diagnosing bugs related to control flow and state.

Acknowledgments. The work presented in this paper has been carried out in context of the VoLL-KI project (grant 16DHKBI091), funded by Bundesministeriums für Bildung und Forschung (BMBF).

References

Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J., 2009. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82, 1780–1792.

Anderson, J.R., 1993. Rules of the mind. Psychology Press.

Bayerkuhnlein, M., Wolter, D., 2024. Model-Based Diagnosis with ASP for Non-groundable Domains, in: International Symposium on Foundations of Information and Knowledge Systems. Springer, pp. 363–380.

Bayerkuhnlein, M., Wolter, D., 2023. Model-Based-Diagnosis for Assistance in Programming Exercises, in: European Conference on Artificial Intelligence. Springer, pp. 459–470.

Chandra, K., Li, T.-M., Nigam, R., Tenenbaum, J., Ragan-Kelley, J., 2024. Watchat: Explaining perplexing programs by debugging mental models. ArXiv Prepr. ArXiv240305334.

Chmiel, R., Loui, M.C., 2004. Debugging: from novice to expert. ACM SIGCSE Bull. 36, 17–21.

Console, L., Friedrich, G., Dupré, D.T., 1993. Model-based diagnosis meets error diagnosis in logic programs, in: Automated and Algorithmic Debugging: First International

Workshop, AADEBUG'93 Linköping, Sweden, May 3–5, 1993 Proceedings 1. Springer, pp. 85–87.

Dovier, A., Formisano, A., Pontelli, E., 2005. A comparison of CLP(FD) and ASP solutions to NP-complete problems, in: Logic Programming: 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005. Proceedings 21. Springer, pp. 67–82.

Forbus, K.D., 1990. The Qualitative Process Engine, in: Weld, D.S., de Kleer, J. (Eds.), Readings in Qualitative Reasoning About Physical Systems. Morgan Kaufmann, pp. 220–235. https://doi.org/10.1016/B978-1-4832-1447-4.50017-1

Johnson, W.L., Soloway, E., 1985. PROUST: Knowledge-based program understanding. IEEE Trans. Softw. Eng. SE-11, 267–275.

Johnson-Laird, P.N., 1989. Mental models, in: Posner, M.I. (Ed.), Foundations of Cognitive Science. The MIT Press, pp. 469–499.

Knobelsdorf, M., Romeike, R., 2008. Creativity as a pathway to computer science, in: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education. pp. 286–290.

Magnani, L., 2011. Abduction, reason and science: Processes of discovery and explanation. Springer Science & Business Media.

Reiter, R., 1987. A theory of diagnosis from first principles. Artif. Intell. 32, 57–95.

Ross, B.J., 1997. Running programs backwards: the logical inversion of imperative computation. Form. Asp. Comput. 9, 331–348.

Shapiro, E.Y., 1982. Algorithmic program debugging. Yale University.

Sharmin, S., 2021. Creativity in CS1: a literature review. ACM Trans. Comput. Educ. TOCE 22, 1–26.

Weiser, M., 1984. Program slicing. IEEE Trans. Softw. Eng. SE-10, 352–357.

Wotawa, F., Dumitru, V.A., 2022. The Java2CSP debugging tool utilizing constraint solving and model-based diagnosis principles, in: International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems. Springer, pp. 543–554.

Wotawa, F., Stumptner, M., Mayer, W., 2002. Model-Based Debugging or How to Diagnose Programs Automatically, in: Hendtlass, T., Ali, M. (Eds.), Developments in Applied Artificial Intelligence. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 746–757.

Zeller, A., Hildebrandt, R., 2002. Simplifying and isolating failure-inducing input. IEEE Trans. Softw. Eng. 28, 183–200.