Model-Based-Diagnosis for Assistance in Programming Exercises

Moritz Bayerkuhnlein(✉) and Diedrich Wolter

University of Bamberg, An der Weberei 5, 96047 Bamberg, Germany
{moritz.bayerkuhnlein,diedrich.wolter}@uni-bamberg.de

Abstract. Implementing AI methods can be an effective way to understand their inner workings, in particular when learners have to locate bugs. However, programming tasks are time consuming and can be extremely challenging for students. In order to provide assistance, code evaluation platforms have been developed that give immediate feedback in the form of discrepancies between expected and actual output for test cases. While such tests clearly indicate wether or not an implementation is faulty, they do not assist learners in locating the fault in their implementation. We propose to diagnose solution attempts, explaining potential faults with respect to abstract behavior. By framing programming tasks as functional models, we can diagnose the underlying concepts of a task and provide feedback. In this paper we focus on abstract data types as a basis for AI algorithms. The diagnosis method described in this paper produces an explanation of a fault in the form of a description based on a reconstruction of hidden program states. Applying the method to student submissions in a programming class shows that the proposed method can effectively identify and locate faults.

Keywords: Model-Based Diagnosis · Fault Localization · Intelligent Tutor System

1 Introduction

The classic AI textbook "Paradigms of AI Programming" by Peter Norvig [16] prominently quotes Alan Perlis stating that learners can only be certain to fully understand an algorithm if they are able to implement – hence also debug – it. We consider programming exercises to be a crucial part in education, helping students to understand and demystify the inner workings of an AI method. Programming tasks operate on multiple levels of abstraction, from the concrete syntax of the programming language to the abstract concepts of the task, all involving a number of *mental models* [13,18].

For algorithms that operate on these abstract representations, debugging can be difficult for learners since a faulty behavior cannot easily be traced back to a

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

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
S. Nowaczyk et al. (Eds.): ECAI 2023 Workshops, CCIS 1948, pp. 459–470, 2024.
https://doi.org/10.1007/978-3-031-50485-3_44

location in the source code. Multiple program components interact in a complex way and faulty intermediate results may only occur under certain conditions and remain hidden in internal program states. While experienced programmers are able to craft decisive test cases that test special cases, students need to learn about such cases first.

Immediate feedback and automated assistance enables novices to learn from a programming task as they face, without overly indulging in handholding the student. This requires automated means to provide feedback for submitted solution attempts, for example using evaluation platforms that run automated tests [10]. Such platforms present an assistance systems to students while developing a solution attempt. Although feedback given in the form of discrepancies between expected and actual output has already been found helpful [11], such approach only assesses correctness of a submitted solution at whole and does not differentiate the abstract representations underlying the task. In other words, existing tools only state whether or not an implementation is faulty. They do not explain *why*, nor do they provide hints to the learner *where* the bug is. We are motivated – also by feedback from our students that suffered interpreting test feedback from existing tools – to develop means for automated feedback that is capable of explaining program faults more intuitively.

We propose to relate faults identified in programming tasks to explicit functional models by means of diagnosis and explicit reconstruction of the faulty system's internal state, providing feedback on level that is close to how underlying concepts are taught. Our approach builds on the common infrastructure of automated test cases, but extends it with model-based diagnosis. This means we do not inspect the actual source code but treat it as a black box, allowing the method to be applied independent of the programming language used. To achieve our aim we propose a model of functional circuits that is related to classical diagnosis domains like electrical circuits. In this paper we consider implementation of Abstract Data Types (ADTs) (e.g., stack, tree, etc.) which underly AI methods (e.g., for managing the fringe in search methods). ADTs are challenging to debug for novices since ADTs are based on information hiding, concealing the internal state.

Figure 1 gives an example of the feedback generated by the method described in this paper for a faulty implementation. In the corresponding programming exercise, a binary search tree had to be implemented. In Sect. 2 we describe how model-based diagnosis can be applied and relate the approach to other techniques for fault localization. Based on a logic model, we apply diagnosis to programs (Sect. 3) and with the help of computational logic tools we then determine a model (Sect. 4) of the faulty behavior that reconstructs hidden states of the program to explain the fault (Sect. 5). As can be seen in Fig. 1, our system produces hypotheses of possible faults, aiming to direct students to bugs in the code. Both, textual and graphical presentations can be provided. In Sect. 5.1 we give first results regarding the effectiveness of the proposed method we obtained for student submissions in an introduction course.

```
1    public void addRec(int key, Node current){
2        if(root.getKey() < current.getKey()){
3            if(current.getLeft() != null){
4                addRec(key,current.getLeft());
5            }else{
6                current.setLeft(new Node(key));
7            }
8        }else if(current.getRight() != null){ //...
```


[Hypothesis 1] **add**-method(s) are faulty, we suspect for example:

example: calling add(1) on tree(nil,2,tree(nil,3,nil)) may have produced tree(nil,2,tree(tree(nil,1,nil),3,nil))

[Hypothesis 2] **inorder and preorder**-method(s) are faulty, we suspect for example:

example: calling inorder() on tree(tree(nil,1,nil),2,tree(nil,3,nil)) may have produced [2,1,3]
example: calling preorder() on tree(tree(nil,1,nil),2,tree(nil,3,nil)) may have produced [2,3,1]

Fig. 1. Excerpt from faulty student code of a binary search tree which always compares the current node's key with the root key, no the provided key (line 2). The resulting diagnosis as text with additional illustration is shown below.

2 Model-Based Diagnosis and Debugging

In this section we discuss approaches to fault localization and show how the problem of localizing faults within a system can be posed as a diagnosis problem, using reasoning from the first principles [9,17], that is, model-based diagnosis (MBD). Following Reiter [17], diagnosis employs a *structural* and a *behavioral* model of a system.

Definition 1 (Diagnosis System). *A diagnosis system consists of* (SD, COMP)*, where* COMP *is a set of components that reside within the system, and the description of the system* SD *defines the behavior of the components in their interaction based on their structure.*

A functionally correct system consists of components that exhibit the behavior of SD. If the system is observed to behave abnormally, i.e. it produces unexpected output, then diagnosis traces back this abnormality to one or more potentially abnormal components $\{\text{AB}(c_1), ..., \text{AB}(c_n)\}$. We define all observable inputs and outputs on terminals of a system as a finite set OBS.

In the consistency-based approach to model-based diagnosis, a component c is admitted to show arbitrary behavior regardless of the specification in SD only if it is declared by $\text{AB}(c)$ to act abnormally [17]. Diagnosis is thus the task of determining minimal sets of components such that their conjectured abnormality explains all observations.

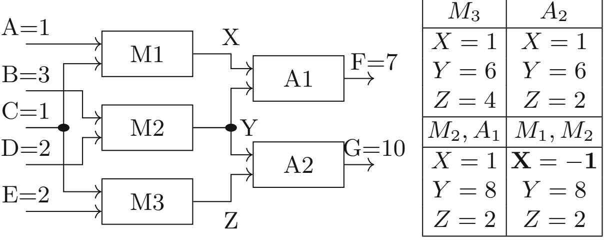
Definition 2 (Diagnosis). *For a system* (SD, COMP) *and* OBS *formalized as logical sentences, a diagnosis is a set* $\Delta \subseteq$ COMP *iff* SD $\cup$ OBS $\cup$ $\{\text{AB}(c) | c \in \Delta\} \cup \{\neg\text{AB}(c) | c \in (\text{COMP} \setminus \Delta)\}$ *is consistent. A diagnosis* Δ *is minimal if there is no alternative diagnosis* $\Delta' \subset \Delta$*.*

2.1 Model-Based Software Debugging

Applying MBD to debugging has been done in the form of Model-Based Software Debugging (MBSD) initially in Logic Programming [7], but has since been adapted to functional [20] and object-oriented paradigms [23]. MBSD derives a system model directly from the source code and the programming language semantics. Statements and expressions constitute set COMP, and, in contrast to MBD, SD does not provide the specification; rather, it mirrors the faulty implementation. Similarly, the role of OBS is flipped, as they now represent the expected output according to a test oracle [21]. MBSD is aimed at an application for experienced programmers [21]. It is generally assumed that the experienced programmer knows what they are doing, bugs are expected to be infrequent and are most likely repairable by slight modifications [21]. In the context of a learning support system, de Barros et al. [3] use structural abstraction and a hierarchical model-based approach to reason about specified code patterns as abstract components, with the goal of establishing a better dialog when communicating errors to students in terms of their problem solving strategies. A challenge when referring to source code (aside from adaption efforts to specific programming languages) lies in the fact that diverse implementations can produce the same behavior using dramatically different techniques.

2.2 Fault Localization

Lately, even machine learning methods have been applied to fault localization and automated repair of code [2]. By contrast, classical models are based on formal specification and testing against a formal specification, hence ensuring correctness of the output. While well-designed complex tests can be very effective to verify correctness of a program, they provide no direct pointers to bugs in the code. Moreover, it requires much care to design a minimal set of test cases that is able to detect all reasonable faults. It is therefore attractive to run many tests in an exhaustive manner. As a downside of exhaustive testing, test output may be overwhelming. A further challenge faced in testing is that a single fault can cause multiple tests to fail. Several methods have been proposed to compile test failures into a ranked set of fault candidates, a prominent example being the family of spectrum-based fault analysis [19] which has also been applied to the challenging task of multiple fault localization [1]. The idea of spectrum-based based methods is to compute a metric that derives the likelihood of a component being faulty from the number of faulty tests the component was used (along other components) in relation to participation in passed tests. While computing such metrics can be done efficiently, they only provide an estimate and rely on an appropriate balancing of the test cases. In contrast to such heuristic methods,



M_3	A_2
$X = 1$	$X = 1$
$Y = 6$	$Y = 6$
$Z = 4$	$Z = 2$
M_2, A_1	M_1, M_2
$X = 1$	$\mathbf{X = -1}$
$Y = 8$	$Y = 8$
$Z = 2$	$Z = 2$

Fig. 2. Functional circuit on a range of [0,15] (4-bit), diagnosed component sets and their context as corresponding variable values.

model-based diagnosis has the advantage of giving correct results and is able to provide justification in form of a logic model.

3 Modelfinding for Diagnosis

The presence of a fault introduces unknown behavior into a system, which manifests itself as symptoms. Diagnosing involves identifying and isolating a cause from the observed symptoms. This form of reasoning is known as abductive reasoning. *Abductive diagnosis*, as defined by Console in [8], uses strong fault models that explicitly model faulty system behavior. By restricting the outcome to a set of possible causes that act as justifications, stronger fault models reduce the number of candidate diagnoses. However, additional modelling effort is required.

Instead, and unlike abductive diagnosis, our approach reconstructs the system state from the observations made. Related approaches to model-based diagnosis from the field of constraint programming have been termed *constructive abduction* [14]. To illustrate the benefit of constructive abduction over consistency-based diagnosis, consider a classic example of a circuit consisting of multiplier components $\{\mathrm{M1}, \mathrm{M2}, \mathrm{M3}\}$ and adder components $\{\mathrm{A1}, \mathrm{A2}\}$, as shown in Fig. 2, where the components perform operations on 4-bit integer values from 0 to 15. Ports A to E are the inputs and ports F, G are the observable system outputs. The system thus computes $F = (A{\cdot}C)+(B{\cdot}D)$ and $G = (B{\cdot}D)+(C{\cdot}E)$. The inputs shown in Fig. 2 indicate a discrepancy, since $G = 10$ differs from the expected value $(3 \cdot 2) + (1 \cdot 2) = 8$.

The diagnoses generated by the consistency-based approach in accordance with Definition 2 are $\Delta = \{\{\mathrm{M3}\}, \{\mathrm{A2}\}, \{\mathrm{M2}, \mathrm{A1}\}, \{\mathrm{M1}, \mathrm{M2}\}\}$, i.e. four possible minimal sets of abnormal components. If we consider the diagnostic system as a system of equations, and the abnormal components output X, Y as *free variables*, we obtain the equations $X + Y = 7$ and $Y + 2 = 10$. We thus have $X + 8 = 7$, which has no solution over the domain of non-negative 4-bit integers, but all other solutions actually provide a justification based on the value assumed to be the output of an abnormal component. In conclusion, diagnosis $\{\mathrm{M1}, \mathrm{M2}\}$ is not a feasible explanation for the observed error. As a side product, we also obtain

values that support a diagnosis, essentially determining a satisfying model or context C such that

$$C \models \text{SD} \cup \text{OBS} \cup \{ab(c) \mid c \in \Delta\} \tag{1}$$

The method can be formulated as a Constraint Satisfaction Problem (CSP), where an assignment of variables is sought from a finite domain, in our case the context C. A practical method can easily be obtained by first-order model finding, using answer set programming (ASP) or SMT solvers.[1]

To find the minimal diagnosis, we use an incremental approach, directly specifying the number of abnormal operations or components we want to diagnose, and incrementally increasing the value; the consistency check by the solver can only consider models that satisfy the exact number of abnormal operations [22].

Once a constellation of abnormal operations is found, the framework excludes it from further diagnosis, so that no super sets are generated. For the scope of this paper we focus on the modelling aspects as the main contribution.

4 Model-Based Diagnosis of Programming Exercises

Different implementations can achieve the same behavior using a variety of underlying mechanisms, yet the behavior follows a common specification. In our work we assume that a specification of the ADT to be implemented is given as an algebraic specification.

Definition 3 (Algebraic Specification). *An algebraic specification (AS) is a tuple* (Σ, E) *where signature* Σ *has a finite set of sorts* S *(i.e. type names) and operations of structure* $op : s_1 \times ... \times s_n \rightarrow s \mid s_i \in S$*. Semantics of op is defined by equations, as a set of axioms* E*.*

A natural level of abstraction for components in the sense of a diagnostic system is then the level of individual methods defined by the ADT and in use in a sequence of operation calls. During diagnosis, we model each operation call as a single component in order to differentiate between different conditions in which a component is in use. For example, a stack implementation is composed of components that implement initialisation, PUSH, POP, etc. Two successive PUSH operations are modelled as two different components PUSH_1, PUSH_2. Component behavior is defined as input and output pairs or functions that pass values, but not all values are observable. For example, the result of a PUSH operation modifies the stack and usually does not return a directly observable value. Consequently, values appearing at the terminals of intermediate components must be reconstructed during diagnostic reasoning. We say that if the output produced by a component c differs from what is specified in a given AS, it behaves abnormally, written $ab(c)$. For observable outputs, abnormality can

[1] For experiments, specifications are formalised in the interactive theorem prover Isabelle/HOL [15] using its integrated model finding capabilities [5] to perform the constructive abduction.

be inferred from observations. For unobservable components, abnormality can only be inferred by reasoning about their behavior within a network of components as a whole. Abnormality of a component is treated as a justification for any output produced by the component, both in conflict with and in accordance with the specification. Since components of the same functional type, e.g. $\textsc{push}_i$, $i = 1, 2, \ldots$, are based on a single implementation, we introduce a rule that propagates an inferred abnormality to all components of that type: $ab(c) \wedge type(c, op_i) \wedge type(c', op_i) \rightarrow ab(c')$.

The goal of diagnosis is to identify (i) a set of individually faulty components, or (ii) a faulty mechanic that may be spread across multiple components and cannot be attributed to a single component. To this end, we use compound statements of the form $a, b \in \textsc{comp}$: $a \wedge b$ to express that either there are problems in both the mechanics implementing a and b, or there is a common mechanic that is broken by both, including side effects of one that affect the other. Analogously, $a \vee b$ denotes an alternative diagnosis that either a or b is abnormal, and occurs whenever some uncertainty remains from the observations made.

4.1 Algebraic Specification as Diagnosis System

Given an AS (Σ, E), we construct test cases by arbitrarily composing the methods of the ADT, i.e. the set of non-primitive functions occurring in op and determining the expected results according to E, for an example see Fig. 3. Each test case is then applied to the code and fully executed to be diagnosed, collecting the observable outputs. Each test case is fully executed, thus the actual output can deviate on multiple occasions from the expected output which aids identifying aftereffects of faults. The method calls occurring in the tests then constitute the

```
definition create:: "nat ⇒ S"  ("create'(_')") where "create N ≡ (empty,N)"
definition push :: "S ⇒ E ⇒ S" ("push'(_,_')") where "push(P,E) ≡ (E on fst P,snd P)"
axiomatization
  size :: "S ⇒ nat"     ("size'(_')") and
  pop :: "S ⇒ S × E"    ("pop'(_')") and
  isEmpty :: "S ⇒ bool" ("isEmpty'(_')") and
  isFull :: "S ⇒ bool"  ("isFull'(_')") and
  capacity :: "S ⇒ nat" ("capacity'(_')")
where
 AA: "capacity(S)<size(S) ⟶ err(S)"  and AB: "err (pop(create(N)))" and
 A1: "fst pop(push(s,e)) = s"          and A2: "snd pop(push(s,e)) = e" and
 A3: "isEmpty(push(s,e)) = False"      and A4: "isEmpty(create(N)) = True" and
 A5: "size(create(N)) = 0"             and A6: "size(push(s,e)) = size(s)+1 " and
 A7: "capacity(create(N)) = N"         and A8: "capacity(push(s,e)) = capacity(s)" and
 A9: "isFull(S) = (size(S)=capacity(S))"
```

Fig. 3. Specification of capacity bounded abstract datatype stack formalized in Isabelle/HOL using definitions on pairs and inductive datatypes for constructors and axiomatization to specify the remaining behavior

set of components COMP_{AS} of the diagnosis system $(\text{COMP}_{AS}, \text{SD}_{AS})$. Behavior representation SD_{AS} and observations OBS_{AS} is given by test cases.
A Diagnosis Δ of SD_{AS} is then: $\text{SD}_{AS} \cup \text{OBS} \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in \text{COMP}_{AS} \setminus \Delta\}$

As opposed to primitive datatypes such as boolean and integer, ADTs perform information hiding. There is usually not even a method to test for equality. So we can only check the tests on the basis of *observable sorts*. This has implications for the representation of values propagated through the components of a diagnostic system, since instead checking instances of non-observable sorts (i.e. ADTs) it is only possible to check whether they are *observational equal*. That is, instances cannot be distinguished by "experiments", as in any sequence of operations that results in an observable sort [4].

As a means of transferring information between the components of COMP_{AS}, we choose to represent values of unobservable sorts as first-order ground terms using the constructors of the ADT (e.g. `node(Key,LeftChild, rightChild)` for a binary tree). Through constructive abduction we effectively reconstruct values for the purpose of justification whenever a feasible diagnosis is found. Using this representation, we anchor any justification we make to the structure and state of the data type. However, this assumes that the observations made can be deterministically reconstructed while relying only on this simplified representation. Whenever the value represented here is the product of a faulty component, we will refer to it as a *corrupt* value or state.

Example 1. Let component POP_i represent the operation $\text{POP} : \textit{stack} \rightarrow \textit{stack} \times \textit{char}$. Provided with input term $t_1 = \text{PUSH}(\text{CREATE}(3), \mathtt{a})$, i.e. a stack of size 3 containing only literal 'a', the output of POP_i, namely $\text{POP}(\text{PUSH}(\text{CREATE}(3), \mathtt{a}))$, can be rewritten as $\text{CREATE}(3)$ using the axiom $\text{POP}(\text{PUSH}(s, e)) = (s, e)$. Assuming POP_i to act normally, one can only infer from the element e returned that the state of the data structure before calling POP_i is consistent with t_1, not that it must be identical to t_1.

5 Inferring Hidden Values from Testcases

Identifying faulty components within a system requires tracing symptoms through the propagation of the components. A key part of this propagation and reconstruction of the state of the system is determined by how we structure and relate the observations, forming a *structural model*. The model of a physical device is defined by the actual physical connections between each component. This is not necessarily the case for a more abstract system. *Connections* between calls to the datatype are the datatype values or states and any other value passed between calls. We consider *components* as the operation calls, where the implementing operation is the type of that component. Values, including the representation of ADT states, are passed from one operation call to the next, usually terminating in an observing operation that returns a primitive value.

If we want to effectively describe and reason about a failure, information hiding must be overcome by reconstructing states. A call to a component flagged as abnormal means that no guarantees can be made about the output of that operation. This induces the possibly of corrupted states through the observed context, i.e. the sequence of calls following the values output by the observers, resulting in a context with (at least) a behaviorally equivalent representation [12]. As shown in Fig. 4 (right), we structure the connection model in a branching fashion in order to relate the information. Where a path through the diagram represents a test, tests share the same history and therefore the same state. Whenever a test sequence branches from another, the values present on both branches must be identical. Following the notation introduced in [6], we formalize these branches using shared variables, so that any constraint imposed on one variable affects all its branches.

Example 2. Expression 2 represents two test sequences that overlap until op_3, where one test, represented by the shared variable **b**, observes and terminates with f, while the other test continues to manipulate the state to c, d and e.

$$\exists\, a\, b\, c\, d\, e\, f.\mathrm{create}_1(3, a) \wedge \mathrm{push}_2(a, 1, \mathbf{b}) \wedge \mathrm{push}_3(\mathbf{b}, 2, c) \wedge \mathrm{pop}_3(\mathbf{b}, f, 1) \wedge \mathrm{pop}_4(c, d, 2) \wedge \mathrm{pop}_5(d, e, 2) \quad (2)$$

Therefore, when generating tests for an implementation, we require a certain amount of overlap in the test cases so that operations operate on the same state according to the structural model. In the case of abstract data type implementations and other API-like specifications, tests can be generated using for example a breadth-first search.

5.1 Constructing Diagnosis

From the consistency-based definition 2, a regular diagnosis result identifies the bug by providing a set of components $\{c \mid c \in \mathrm{COMP} \wedge ab(c)\}$ that need to be repaired. In our case, this set includes a set of operations from the specification that must deviate from the specification. An operation may implement multiple cases, or have behavior that depends on a particular range of values, in which case it is helpful to contextualize the diagnosis. As mentioned in Sect. 3 using the reconstructed values, a diagnosis can also be justified when faced with a discrepancy based on the inferred input and output values.

Fig. 4 (left) illustrates Example 2 as a component connection model. On an operation-call level, i.e. distinguishing between the individual calls during diagnosis, we obtain the (4) minimal diagnoses: The minimal diagnoses are searched for incrementally following [22] by allowing only a fixed number of abnormal components.
(i) $ab(pop_5)$, justified by $d = push(create(3), 1)$ but returning 2. (ii) $ab(pop_4)$, justified by $c = push(push(create(3), 1), 2)$ producing $d = push(_, 2)$[2]. (iii) $ab(push_3)$, justified by $b = push(create(3), 1)$ but returning $c =$

[2] Wildcard (_) denotes assignment where no context constrains the value.

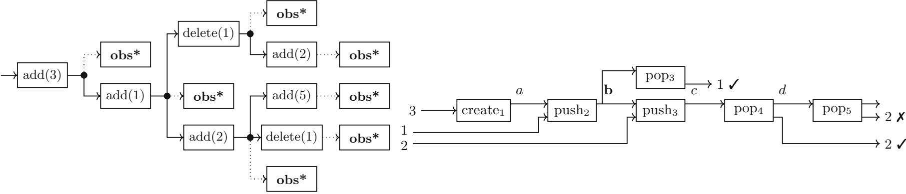


Fig. 4. Visual Representation of test cases as circuits. Binary Search Tree test cases represented as a branching functional circuit (left). Sequence of stack operations annotated with observations (right).

$push(push(_, 2), 2)$. (iv) $ab(push_2)$, justified by $a = create(3)$ but producing $b = push(_, 2)$ and $ab(pop_3)$ justified by $b = push(_, 2)$ but returning 1.

Qualitatively, these justifications can now be scrutinized to assess their plausibility. For example, just by aggregating and looking at the diagnosis operation by operation, as described in Sect. 4, we can check whether reconstructed values still induce deterministic functions from their inputs and outputs.

In the following, we demonstrate the quality of diagnosis that the approach can provide. For demonstration, we modeled a programming task to implement an ADT stack and implemented the approach. We asked students in a first-year introductory course on algorithms and data structures at our university to implement the stack based on an array in Java at the beginning of the semester as part of their homework. The submitted code was stored and evaluated using the INGInious platform [10]. 74 students submitted multiple attempts to solve their homework. In total, 267 submissions were recorded, of which 49 had solution errors in their stack implementation, detected by manually written test cases and the INGInious platform. We demonstrate the use of the approach by localiszing the incorrect behavior of the implementations.

Table 1 shows the types of erroneous implementations that we identified by manually scanning all submissions. The table also shows the corresponding diagnosis found by the proposed method, based on testing against a stack using an exhaustive test suite. Note that the correctness of a diagnosis here depends on the coverage of the test suite. The diagnosis found for the faulty implementations is correct, i.e. the diagnosis covers the actual faults in the implementation, although there are false positives among the conjunctions. However, these can be checked using the reconstructed values. For example, if there are several possible values for a variable, the justification is not deterministic and therefore not plausible.

We can use a reduced test set to get good, potentially equivalent diagnoses. As shown in the results obtained for a binary tree datatype implementation of Table 1. Where the implementations have only been tested on the circuit shown in Fig. 4 (left). However, this does affect the ability to effectively reconstruct values as the presence of the constraint decreases. The ground-truths are found by the diagnosis and here listed first.

Table 1. Observed errors in the student submission for an array-based stack data type and binary search tree. For all error classes, the diagnosis covered the actual errors in the code. (*)-Asterisk marks conjunctions that suggested an implausible justification based on non-determinism.

ID	Description	Diagnosis Δ		
$\mathbf{ar}_1$:	Array not initialized	$create$	$\vee$	$push$
$\mathbf{ar}_2$:	Array indexing starts at 1	$push$	$\vee$	$(create \wedge full)$
$\mathbf{ie}_1$:	empty is inverted	$empty$	$\vee$	$(create \wedge push \wedge pop)^*$
$\mathbf{if}_1$:	full is inverted	$full$	$\vee$	$(create \wedge push \wedge pop)^*$
$\mathbf{ieif}_1$:	both empty and full are inverted	$(empty \wedge full)$	$\vee$	$(create \wedge push \wedge pop)^*$
$\mathbf{po}_1$:	pop on empty returns constant	pop	$\vee$	$(create \wedge push \wedge empty)^*$
$\mathbf{po}_2$:	pop always returns constant	pop		
$\mathbf{po}_3$:	pop only observes	pop	$\vee$	$(push \wedge full)^*$
$\mathbf{pu}_1$:	push on full no exception	$push$	$\vee$	$(pop \wedge empty \wedge full)^*$
$\mathbf{pu}_2$:	push on full overwrites top element	$push$	$\vee$	$(pop \wedge empty \wedge full)^*$
$\mathbf{lst}_1$	tree is list structured	add	$\vee$	$(find \wedge inorder)$
$\mathbf{dl}_1$	delete has no effect	$delete$		
$\mathbf{dl}_2$	delete removes subtree	$delete$	$\vee$	$(add \wedge find \wedge inorder)$
$\mathbf{ip}_1$	in- implements postorder	$inorder$	$\vee$	$(add \wedge find)$
$\mathbf{co}_1$	exception if not contained	$find$		

6 Conclusion and Future Work

As an effort to provide more informative automated feedback for student programming tasks, we design a computational logic model in an adaption of model-based diagnosis. The model presented here is able to isolate faults and reason explicitly about the internal state of the faulty system by means of constructive abduction. The approach bridges automated testing to abstract reasoning, allowing intuitive explanations of a fault.

In future work we want to investigate diagnosis of complex AI algorithms. This poses challenges with respect to efficiency of the diagnosis and with respect to handling components that influence the control flow of a program. Last but not least, we plan to conduct a user study to learn about most suitable levels of abstractions when communicating a fault to the student.

References

1. Abreu, R., Zoeteweij, P., Gemund, A.J.V.: Spectrum-based multiple fault localization. In: Proceedings of IEEE/ACM International Conference on Automated Software Engineering, pp. 88–99. IEEE (2009)
2. Allamanis, M., Jackson-Flux, H., Brockschmidt, M.: Self-supervised bug detection and repair. In: Proceedings of 35th Conference on Neural Information Processing Systems (NeurIPS 2021) (2021)
3. de Barros, L.N., Pinheiro, W.R., Delgado, K.V.: Learning to program using hierarchical model-based debugging. Appl. Intell. **43**(3), 544–563 (2015)
4. Bidoit, M., Hennicker, R., Wirsing, M.: Behavioural and abstractor specifications. Sci. Comput. Program. **25**(2–3), 149–186 (1995)

5. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_11
6. Camilleri, A., Gordon, M., Melham, T.: Hardware verification using higher-order logic. University of Cambridge, Computer Laboratory, Technical report (1986)
7. Console, L., Friedrich, G., Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In: Fritzson, P.A. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 85–87. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0019402
8. Console, L., Torasso, P.: A spectrum of logical definitions of model-based diagnosis 1. Comput. Intell. **7**(3), 133–141 (1991)
9. Davis, R.: Diagnostic reasoning based on structure and behavior. Artif. Intell. **24**(1–3), 347–410 (1984)
10. Derval, G., Gego, A., Reinbold, P., Frantzen, B., Van Roy, P.: Automatic grading of programming exercises in a MOOC using the INGinious platform. In: European Stakeholder Summit on Experiences and Best Practices in and Around MOOCs (EMOOCS 2015), pp. 86–91 (2015)
11. Hao, Q., et al.: Towards understanding the effective design of automated formative feedback for programming assignments. Comput. Sci. Educ. **32**(1), 105–127 (2022)
12. Hennicker, R.: Context induction: a proof principle for behavioural abstractions and algebraic implementations. Formal Aspects Comput. **3**, 326–345 (1991)
13. Johnson-Laird, P.N.: Mental Models. MIT Press, Cambridge (1989)
14. Ligeza, A., et al.: Constraint programming for constructive abduction. A case study in diagnostic model-based reasoning. In: Kościelny, J.M., Syfert, M., Sztyber, A. (eds.) DPS 2017. AISC, vol. 635, pp. 94–105. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-64474-5_8
15. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
16. Norvig, P.: Paradigms of AI Programming: Case Studies in Common Lisp. Morgan Kaufmann (1992)
17. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. **32**(1), 57–95 (1987)
18. Robins, A., Rountree, J., Rountree, N.: Learning and teaching programming: a review and discussion. Comput. Sci. Educ. **13**(2), 137–172 (2003)
19. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges. Technical report. arXiv:1607.04347 (2016)
20. Stumptner, M., Wotawa, F.: Debugging functional programs. In: IJCAI, vol. 99, pp. 1074–1079. Citeseer (1999)
21. Wieland, D.: Model-based Debugging of Java Programs using Dependencies. Ph.D. thesis, Technische Universität Wien (2001)
22. Wotawa, F., Kaufmann, D.: Model-based reasoning using answer set programming. Appl. Intell. **52**, 1–19 (2022)
23. Wotawa, F., Stumptner, M., Mayer, W.: Model-based debugging or how to diagnose programs automatically. In: Hendtlass, T., Ali, M. (eds.) IEA/AIE 2002. LNCS (LNAI), vol. 2358, pp. 746–757. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-48035-8_72