

# Program Analysis and Manipulation to Reproduce Learners' Erroneous Reasoning

Claus Zinn

Department of Computer Science  
University of Konstanz  
Box D188, 78457 Konstanz, Germany  
`claus.zinn@uni-konstanz.de`

**Abstract.** Pedagogical research shows that learner errors are seldom random but systematic. Good teachers are capable of inferring from learners' input the erroneous procedure they are following, and use the result of such deep cognitive diagnoses to repair its incorrect parts. We report a method for the automatic reconstruction of such erroneous procedures based on learner input and the analysis and manipulation of logic programs. The method relies on an iterative application of two algorithms: an innovative use of algorithmic debugging to identify learner errors by the analysis of (initially) correct (*sic*) Prolog-based procedures, and a subsequent program manipulation phase where errors are introduced into (initially) correct procedures. The iteration terminates with the derivation of an erroneous procedure that was followed by the learner. The procedure, and its step-wise reconstruction, can then be used to inform remedial feedback.

## 1 Introduction

The diagnosis of learner input is central for the provision of effective scaffolding and remedial feedback in intelligent tutoring systems. A main insight is that errors are rarely random, but systematic. Either learners have acquired an erroneous procedure, which they execute in a correct manner, or learners attempt to execute a correct procedure but encounter an *impasse* when executing one of its steps. To address the impasse, learners are known to follow a small set of *repair* strategies such as skipping the step in question, or backing-up to a previous decision point where performing the step can be (erroneously) avoided.

Effective teaching depends on deep cognitive diagnosis of such learner behaviour to identify erroneous procedures or learners' difficulties with executing correct ones. State-of-the-art intelligent tutoring systems, however, fail to give a full account of learners' erroneous skills. In *model tracing* tutors (*e.g.*, the Lisp tutor [2]; the Algebra Tutor [4]), appropriately designed user interfaces and tutor questions invite learners to provide their answers in a piecemeal fashion. It is no longer necessary to reproduce a student's line of reasoning from question to (final) answer; only the student's next step towards a solution is analyzed,

and immediate feedback is given. Model tracing tutors thus keep learners close to ideal problem solving paths, hence preventing learners to fully exhibit erroneous behaviour. *Constraint-based tutors* (e.g., the SQL tutor [7]) perform student modelling based on constraints [9]. Here, diagnostic information is not derived from an analysis of learner actions but of problem states the student arrived at. With no representation of actions, the constraint-based approach makes it hard to identify and distinguish between the various (potentially erroneous) procedures learners follow to tackle a given problem.

None of the two approaches attempt to explain buggy knowledge or skills. There is no explicit and machine-readable representation to mark deviations of an expert rule from the buggy skill; and also, there is no mechanism for automatically deriving buggy skills from correct ones. In this paper, we report a method capable of reconstructing erroneous procedures from expert ones. The method is based on an iterative analysis and manipulation of logic programs. It relies on an innovative use of algorithmic debugging to identify learner error by the analysis of (initially) correct (*sic*) Prolog-based procedures (modelling expert skills), and a subsequent program manipulation to introduce errors into the correct procedure to finally produce the erroneous procedure followed by the learner. The method extends our previous work [14] by having algorithmic debugging now qualify the irreducible disagreement with its cause, *i.e.*, by specifying those elements in the learner's solution that are missing, incorrect, or superfluous. Moreover, we have defined and implemented a perturbation algorithm that can use the new information to transform Prolog programs into ones that can reproduce the observed error causes.

The remainder of the paper is structured as follows. Sect. 2 introduces the domain of instruction (multi-column subtraction), typical errors and how they manifest themselves in this domain, and our previous work on adapting Shapiro's algorithmic debugging to support diagnosis in intelligent tutoring systems. Sect. 3 first describes our new extension to Shapiro's technique; it then explains how we interleave the extended form of algorithmic debugging with automatic code perturbation, and how this method can be used iteratively to track complex erroneous behaviour. In Sect. 4, we give examples to illustrate the effectiveness of the method. In Sect. 5, we discuss related work, and Sect. 6 lists future work and concludes.<sup>1</sup>

## 2 Background

Our approach to cognitive diagnosis of learner input is applicable for any kind of domain that can be encoded as a logic program. For illustration purposes, we focus on the well-studied example domain of multi-column subtraction.

---

<sup>1</sup> For our research, we have chosen basic school arithmetics, a subject that most elementary schools teach in 3rd. and 4th. grade. In the paper, we use the terms "pupil", "learner" and "student" to refer to a school child.

```

subtract(PartialSum, Sum) :-
    length(PartialSum, LSum),
    mc_subtract(LSum, PartialSum, Sum).

mc_subtract(_, [], []).
mc_subtract(CurrentColumn, Sum, NewSum) :-
    process_column(CurrentColumn, Sum, Sum1),
    shift_left(CurrentColumn, Sum1, Sum2, ProcessedColumn),
    CurrentColumn1 is CurrentColumn - 1,
    mc_subtract(CurrentColumn1, Sum2, SumFinal),
    append(SumFinal, [ProcessedColumn], NewSum).

process_column(CurrentColumn, Sum, NewSum) :-
    last(Sum, LastColumn),      allbutlast(Sum, RestSum),
    subtrahend(LastColumn, S),  minuend(LastColumn, M),
    S > M, !,
    add_ten_to_minuend(CurrentColumn, M, M10),
    CurrentColumn1 is CurrentColumn - 1,
    decrement(CurrentColumn1, RestSum, NewRestSum),
    take_difference(CurrentColumn, M10, S, R),
    append(NewRestSum, [(M10, S, R)], NewSum).

process_column(CurrentColumn, Sum, NewSum) :-
    last(Sum, LastColumn),      allbutlast(Sum, RestSum),
    subtrahend(LastColumn, S),  minuend(LastColumn, M),
    % S =< M,
    take_difference(CurrentColumn, M, S, R),
    append(RestSum, [(M, S, R)], NewSum).

shift_left( _CurrentColumn, SumList, RestSumList, Item ) :-
    allbutlast(SumList, RestSumList),
    last(SumList, Item).

add_ten_to_minuend( _CC, M, M10) :- irreducible, M10 is M + 10.
take_difference(    _CC, M, S, R) :- irreducible, R is M - S.

decrement(CurrentColumn, Sum, NewSum) :- irreducible,
    last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
    M == 0, !,
    CurrentColumn1 is CurrentColumn - 1,
    decrement(CurrentColumn1, RestSum, NewRestSum),
    NM is M + 10, NM1 is NM - 1,
    append( NewRestSum, [ (NM1, S, R) ], NewSum).

decrement(CurrentColumn, Sum, NewSum) :- irreducible,
    last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
    % \+ (M == 0),
    M1 is M - 1,
    append( RestSum, [ ( M1, S, R ) ], NewSum ).

minuend( (M, _S, _R), M).      subtrahend( (_M, S, _R), S).      irreducible.

```

Fig. 1. Multi-column subtraction in Prolog

## 2.1 Multi-column Subtraction

Fig. 1 gives an implementation of multi-column subtraction in Prolog. Sums are processed column by column, from right to left. The predicate `subtract/2` determines the sum's length and passes the arguments to `mc_subtract/3`, which implements the recursion.<sup>2</sup> The predicate `process_column/3` gets a partial sum, processes its right-most column and takes care of borrowing (`add_ten_to_minuend/3`) and pay-back (`decrement/3`) actions. Sums are encoded as Prolog lists of columns, where a column is represented as 3-element term (`M`, `S`, `R`) representing minuend, subtrahend, and result cell.<sup>3</sup> The program code implements the *decomposition method*. If the subtrahend `S` is larger than the minuend `M`, then `M` is increased by 10 (borrow) before the difference between `M` and `S` is taken. To compensate, the minuend in the column left to the current one is decreased by one (pay-back).

## 2.2 Error Analysis in Multi-column Subtraction

While some errors may be caused by a simple oversight (usually, pupils are able to correct such errors as soon as they see them), most errors are *systematic* errors (those keep re-occurring again and again). It is the systematic errors that we aim at diagnosing as they indicate a pupil's wrong understanding about some subject matter. Fig. 2 lists the top four errors of the Southbay study. [11, Chapter 10]; the use of squares is explained below.

$\begin{array}{r} 5 \quad 2 \quad \square 4 \\ - 2 \quad 9 \quad 8 \\ \hline = 3 \quad 7 \quad 4 \end{array}$	$\begin{array}{r} 3 \quad \square \\ \mathbf{4} \quad 1 \quad 0 \quad 14 \\ - 1 \quad 8 \quad 7 \\ \hline = 2 \quad 2 \quad 7 \end{array}$	$\begin{array}{r} 7 \\ \mathbf{8} \quad \square \\ \mathbf{9} \quad 10 \quad 14 \\ - 2 \quad 3 \quad 7 \\ \hline = 5 \quad 7 \quad 7 \end{array}$	$\begin{array}{r} \square \\ 3 \quad 12 \\ - 1 \quad 7 \\ \hline = 2 \quad 5 \end{array}$
(a) smaller-from-larger	(b) stops-borrow-at-zero	(c) borrow-across-zero	(d) borrow-decrement

**Fig. 2.** The top four most frequent subtraction errors, see [11]

Errors can be classified according to the following scheme: *errors of omission* – forget to do something; *errors of commission* – doing the task incorrectly; and *sequence errors* – doing the task not in the right order. These errors occur because learners may have acquired an incorrect procedure, or they “know” the correct procedure but are unable to execute it. In [11], VanLehn explains the origin of errors in terms of *impasses* and *repairs*. According to his theory, an impasse occurs when an individual fails to execute a step in a procedure; to overcome the impasse, a meta-level activity – a repair action – is performed

<sup>2</sup> Note that the variable `CurrentColumn` is not required to specify our algorithm for multi-column subtraction; it is rather necessary for the mechanisation of the Oracle (see below), and thus passed on as argument to most of the predicates.

<sup>3</sup> The predicates `append/3` (append two lists), `last/2` (get last list element), and `allbutlast/2` (get all list elements but the last) are defined as usual.

$$\begin{array}{r}
 4\ 6\ 3 \\
 -\ 2\ 5\ 1 \\
 \hline
 =
 \end{array}
 \quad
 \begin{array}{r}
 8\ 3\ 5 \\
 -\ 2\ 1\ 8 \\
 \hline
 =
 \end{array}
 \quad
 \begin{array}{r}
 7\ 0\ 2 \\
 -\ 3\ 4\ 7 \\
 \hline
 =
 \end{array}
 \quad
 \begin{array}{r}
 8\ 0\ 7 \\
 -\ 2\ 0\ 8 \\
 \hline
 =
 \end{array}
 \quad
 \begin{array}{r}
 6\ 0\ 4 \\
 -\ \ \ 8\ 7 \\
 \hline
 =
 \end{array}$$

(a)                      (b)                      (c)                      (d)                      (e)

**Fig. 3.** Exercises on multi-column subtraction [5, page 59f]

to cope with the impasse. While some repairs result in correct outcomes, others result in buggy procedures. Consider the error exhibited in Fig. 2(a). In VanLehn’s theory, this can be explained as follows. The learner cannot perform a complete borrow action; to cope with this impasse, a repair is executed by “backing-up” to a situation where no borrowing is required: the smaller number is always subtracted from the larger number. In Fig. 2(d), the same impasse is encountered with a different repair. The borrow is executed, with correct differences computed for all columns, but the corresponding decrements (paybacks) are omitted. In Fig. 2(b,c), the learner does not know how to borrow from zero. In (b), this is repaired by a “no-op”, that is, a *null operation* with no effect: for minuends equal to zero, the decrement operation is simply omitted (note that the learner performed a correct borrow/payback pair in the other columns). In (c), the learner repairs the situation by borrowing from the hundreds rather than the tens column; this can be modelled as “partial no-op”.

An accurate diagnosis of a learner’s competences can only be obtained when the learner’s performance is observed and studied across a well-designed set of exercises demanding the full set of domain skills. Fig. 3 depicts exercises taken from a German textbook on third grade school mathematics [5]. While none of the errors given in Fig. 2 show-up in the exercise Fig. 3(a), the errors **borrow-no-decrement** and **smaller-from-larger** are observable in each of the tests Fig. 3(b)–3(e). The exercises Fig. 3(c)–3(e) can be used to check for the bugs **stops-borrow-at-zero** or **borrow-across-zero**, but also for others not listed here.

In the sequel, given a learner’s performance across a set of exercises, we aim to diagnose his errors by reconstructing their underlying erroneous procedures *automatically* using algorithmic debugging and program transformation techniques. Our innovative method relies only on expert programs and learner answers.

### 2.3 Shapiro’s Algorithmic Debugging

Shapiro’s algorithmic debugging technique for logic programming prescribes a systematic manner to identify bugs in programs [10]. In the top-down variant, the program is traversed from the goal clause downwards. At each step during the traversal of the program’s AND/OR tree, the programmer is taking the role of the *oracle*, and answers whether the currently processed goal holds or not. If the oracle and the buggy program agree on a goal, then algorithmic debugging passes to the next goal on the goal stack. If the oracle and the buggy program disagree on the result of a goal, then this goal is inspected further. Eventually an *irreducible disagreement* will be encountered, hence locating the program’s clause

where the buggy behavior is originating from. Shapiro’s algorithmic debugging method extends, thus, a simple meta-interpreter for logic programs.

Shapiro devised algorithmic debugging to systematically identify bugs in incorrect programs. Our Prolog code for multi-column subtraction in Fig. 1, however, presents the expert model, that is, a presumably correct program. Given that cognitive modelling seeks to reconstruct students’ erroneous procedure by an analysis of their problem-solving behavior, it is hard to see – at least at first sight – how algorithmic debugging might be applicable in this context. There is, however, a simple, almost magical trick, which we first reported in [14]. Shapiro’s algorithm can be turned on its head: instead of having the Oracle specifying how the assumed incorrect program should behave, we take the expert program to take the role of the buggy program, and the role of the Oracle is filled by students’ potentially erroneous answers. An irreducible disagreement between program behavior and student answer then pinpoints students’ potential misconception(s).

An adapted version of algorithmic debugging for the tutoring context is given in [14]. Students can “debug” the expert program in a top-down fashion. Only clauses declared as being *on the discussion table* are subjected to Oracle questioning so that rather technical steps (such as `last/2` and `allbutlast/2`) do not need to be discussed with learners. Moreover, a Prolog clause whose body starts with the subgoal `irreducible/2` is subjected to Oracle questioning; but when program and Oracle disagree on such a clause, this disagreement is irreducible so that the clause’s (remaining) body is not traversed. In [14], we have also shown how we can relieve learners from answering questions. The answers to all questions posed by algorithmic debugging are automatically reconstructed from pupils’ exercise sheets, given their solution to a subtraction problem. With the mechanisation of the Oracle, diagnoses are obtained automatically.

### 3 Combining Algorithmic Debugging with Program Manipulation

Our goal is to reconstruct learners’ erroneous procedures by only making use of an expert program (encoding the skills to be learned) and learners’ performances when solving exercises. For this, the expert program is transformed in an iterative manner until a buggy program is obtained that reproduces all observed behaviour. The program transformation is informed by the results of algorithmic debugging as the source of the irreducible disagreement points to the part in the program that requires perturbation. To better inform program transformation, we augment algorithmic debugging to also return the nature of the disagreement.

#### 3.1 Causes of Disagreements

Whenever expert and learner disagree on a goal, one of the following cases holds:

- the learner solution *misses* parts that are present in the expert solution, *e.g.*, a result cell in the multi-column subtraction table has not been filled out;

- the learner solution has *incorrect* parts with regard to the expert solution, *e.g.*, a result cell is given a value, albeit an incorrect one; and
- the learner solution has *superfluous* parts not present in the expert solution, *e.g.*, the learner performed a borrowing operation that was not necessary.

We have extended our variant of algorithmic debugging, and its cooperating mechanised Oracle, to enrich irreducible disagreements with their nature (*missing*, *incorrect*, or *superfluous*). Moreover, we have extended algorithmic debugging to record all agreements until irreducible disagreements are encountered. The program perturbation code has access to these past learner performances and uses this information to steer its perturbation strategy.

### 3.2 Main Algorithm

Fig. 4 gives a high-level view of the algorithm for the reconstruction of learners' erroneous procedure. The function `ReconstructErroneousProcedure/3` is recursively called until a program is obtained that reproduces learner behaviour, in which case there are no further disagreements. Note that multiple perturbations may be required to reproduce single bugs, and that multiple bugs are tackled by iterative applications of algorithmic debugging and code perturbation.

```

1: function RECONSTRUCTERRONEOUSPROCEDURE(Program, Problem, Solution)
2:   (Disagr, Cause) ← AlgorithmicDebugging(Program, Problem, Solution)
3:   if Disagr = nil then
4:     return Program
5:   else
6:     NewProgram ← PERTURBATION(Program, Disagr, Cause)
7:     RECONSTRUCTERRONEOUSPROCEDURE(NewProgram, Problem, Solution)
8:   end if
9: end function

10: function PERTURBATION(Program, Clause, Cause)
11:   return chooseOneOf(CAUSE)
12:     DELETECALLTOCLAUSE(Program, Clause)
13:     DELETESUBGOALOFCLAUSE(Program, Clause)
14:     SHADOWCLAUSE(Program, Clause)
15:     SWAPCLAUSEARGUMENTS(Program, Clause)
16: end function

```

**Fig. 4.** Pseudo-code: compute variant of *Program* to reproduce a learner's *Solution*

The irreducible disagreement resulting from the algorithmic debugging phase locates the code pieces where perturbations must take place; its cause determines the kind of perturbation. The function `Perturbation/3` can invoke various kinds of transformations: the deletion of a call to the clause in question, or the deletion of one of its subgoals, or the shadowing of the clause in question by a more specialized instance, or the swapping of the clause' arguments. These perturbations reproduce errors of omission and commission, and reflect the repair strategies

learners use when encountering an impasse. Future transformations may involve the consistent change of recursion operators (reproducing sequence errors), and the insertion of newly created subgoals to extend the body of the clause in question (reproducing irreducible disagreements with cause *superfluous*).

The generic program transformations that we have implemented require an annotation of the expert program clauses with *mode* annotations, marking their arguments as input and output arguments. Our algorithm for clause call deletion, for instance, traverses a given program until it identifies a clause whose body contains the clause in question; once identified, it removes the clause in question from the body and replaces all occurrences of its output argument by its input argument in the adjacent subgoals as well as in the clause’s head, if present. Then, `DeleteCallToClause/2` returns the modified program.

### 3.3 Problem Sets

A befitting diagnosis of a learner’s knowledge and potential misconceptions requires an analysis of his actions across a set of exercises. Our code for algorithmic debugging and code perturbation is thus called for each exercise of a test set. We start the diagnosis process with the expert program; the program will get perturbed whenever there is a disagreement between program and learner behaviour. At the end of the test set, we obtain a program that mirrors learner performance across all exercises. As we will see, some perturbations might seem ad hoc; here, the program is perturbed with base clauses that only reproduce behavior specific to a given exercise. A post-processing step is thus invoked to generalize over exercise-specific perturbations.

Note that we expect learners to exhibit a problem-solving behaviour that is consistent across exercises. To simulate this idealized behaviour – and to test our approach – we have hand-coded buggy variations of the multi-column subtraction routine to reproduce the top-ten errors of the Southbay study [11, Chapter 10]. Consistent learner behaviour is thus simulated by executing these buggy routines. We use the following “driver” predicate to diagnose learners:

```

1: Program ← ExpertProcedure
2: for all Problem of the test set do
3:   Solution ← BuggyProcedure(Problem)
4:   Program ← ReconstructErroneousProcedure(Program, Problem, Solution)
   ▷ side effect: agreements and disagreements are recorded and used.
5: end for
6: GeneralisedProgram ← Generalise(Program)

```

## 4 Examples

We now explain our approach for the typical errors given in Fig. 2.

### 4.1 Simple Omission Error: Omit Call to Clause in Question

In Fig. 2(d), the learner forgets to honor the pay-back operation, following the borrowing that happened in the first (right-most) column. The execution of the

adapted version of algorithmic debugging produces the following dialogue (with all questions automatically answered by the mechanised Oracle):

```

algorithmic_debugging(subtract([(3, 1, S1), ( 2, 7, S2)],
                               [(3, 1, 2), (12, 7, 5)],
                               IrreducibleDisagreement).

do you agree that the following goal holds:
  subtract( [(3, 1, R1), ( 2, 7, R2)],
            [(2, 1, 1), (12, 7, 5)])
|: no.

do you agree that the following goal holds:
  mc_subtract(2, [(3, 1, R1), ( 2, 7, R2)],
              [(2, 1, 1), (12, 7, 5)])
|: no.

  process_column(2, [(3, 1, R1), ( 2, 7, R2)], [(2, 1, R1), (12, 7, 5)])
|: no.

  add_ten_to_minuend(2, 2, 12)
|: yes.

  decrement(1, [( 3, 1, R1)], [( 2, 1, R1)])
|: no.
=> IrreducibleDisagreement=( decrement(1, [ (3,1, R1), (2,1, R1) ]),
                             missing )

```

With the indication of error (the location is marked by  $\square$  in Fig. 2(d)), program transformation now attempts various manipulations to modify the expert program. Given the cause “missing”, the perturbation heuristics chooses to delete the call to the indicated program clause, which succeeds: in the body of the first clause of `process_column/3`, we eliminate its subgoal `decrement(CurrentColumn1, RestSum, NewRestSum)` and subsequently replace its output variable `NewRestSum` with its input variable `RestSum`.<sup>4</sup> With this program manipulation, we achieve the intended effect; the resulting buggy program reproduces the learner’s answer; both program and learner agree on the top `subtract/2` goal.

## 4.2 Complex Error of Omission and Commission

An iterative execution of algorithmic debugging and program manipulation to the problem in Fig. 2(a) shows how a complex error is attacked step by step.

*First Run.* Running algorithmic debugging on the expert program and the learner’s solution concludes the dialogue (now omitted) with the disagreement

```
add_ten_to_minuend( 3, 4, 14)
```

<sup>4</sup> The subgoal introducing `CurrentColumn1` becomes obsolete, and is removed too.

and the cause “missing”. The code perturbation algorithm makes use of `DeleteCallToClause/2` to delete the subgoal `add_ten_to_minuend/3` from the first program clause of `process_column/3`; the single occurrence of its output variable `M10` is replaced by its input variable `M` in the subsequent calls to `take_difference/4` and `append/3`.

*Second Run.* Algorithmically debugging the modified program yields

```
decrement(2, [(5, 2, R3), (2, 9, R2)], [(5, 2, R3), (1, 9, R2)])
```

a disagreement with cause “missing”. Again, `DeleteCallToClause/2` is invoked, now to delete the subgoal `decrement/3` from the first clause of `process_column/3`. The occurrence of its output variable `NewRestSum` in the subsequent call to `append/3` is replaced by its `decrement/3` input variable `RestSum`; also the subgoal introducing `CurrentColumn1` is deleted. With these changes, we obtain a new program that is closer in modelling the learner’s erroneous behaviour.

*Third Run.* Re-entering algorithmic debugging with the modified expert program now yields an irreducible agreement in the ones column with cause “incorrect”:

```
take_difference(3, 4, 8, 4)
```

A mere deletion of a call to the clause in question is a bad heuristic as the result cell must obtain a value; moreover, past learner performances on the same problem test set indicate that the learner successfully executed instances of the skill `take_difference/4`. We must thus perturbate the clause’s body, or shadow the clause with an additional, more specialized clause. A simple manipulation to any of the clause’s subgoals fails to achieve the intended effect. To accommodate the result provided by the learner, we shadow the existing clause with:

```
take_difference(_CC, 4, 8, 4) :- irreducible.
```

Note that this new clause can be taken directly from the Oracle’s analysis. While the new clause covers the learner’s input, it is rather specific.

*Fourth Run.* We now obtain an irreducible disagreement in the tens column:

```
take_difference(2, 2, 9, 7)
```

with cause “incorrect”. Similar to the previous run, we add another clause for `take_difference/4` to capture the learner’s input:

```
take_difference(_CC, 2, 9, 7) :- irreducible.
```

With these changes to the expert program, we now obtain a buggy program that entirely reproduces the learner’s solution in Fig. 2(a).

*Generalization.* When we run the driver predicate with the buggy learner simulation “smaller-from-larger” for the exercises given in Fig. 3, we obtain a buggy program which includes the following base clauses for `take_difference/4`:

```
take_difference(3, 5,8,3).      take_difference(3, 2,7,5).
take_difference(2, 0,4,4).      take_difference(2, 7,8,1).
take_difference(2, 4,7,3).      take_difference(2, 0,8,8).
```

complementing the existing general clause (see Fig. 1).

We use the inductive logic program Progol [8] to yield a more general definition for taking differences. For this, we construct an input file for Progol in which we re-use the mode declarations used to annotate the expert program. Progol’s background knowledge is defined in terms of the subtraction code (see Fig. 1), and Progol’s positive examples are the base cases given above. Moreover, the expert’s view on the last two irreducible disagreements, namely, `take_difference(4,8,-4)` and `take_difference(2,9,-7)` are taken as negative examples to support the generalisation of the learner’s `take_difference/3` clause.<sup>5</sup>

Running Progol (version 4.2) on the input file yields a new first clause of taking differences that generalizes its base cases. Taken together, we obtain:

```
take_difference(A,B,C) :- leq(A,B), sub(B,A,C).
take_difference(A,B,C) :- irreducible, C is A-B.
```

### 4.3 Error Analysis: Stops-Borrow-at-Zero and Borrow-across-Zero

The solutions given in Fig. 2(b) and Fig. 2(c) indicate two different repair strategies when encountering the impasse “unable to borrow from zero”. We give a detailed discussion for the first, and an abridged one for the second error type.

**Error: Stops-Borrow-at-Zero.** A first run of algorithmic debugging returns the following irreducible disagreement in the tens column with the cause “incorrect”:

```
decrement(2, [ (4, 1, R1), (0, 8, R2)], [ (3, 1, R1), (9, 8, R2)]).
```

Note that the learner is executing the `decrement/3` operation successfully in the hundreds (and in prior exercises with minuends different from zero). A program perturbation other than the deletion of the call to `decrement/3` – as in the previous two examples – is needed. The clause `decrement/3` must thus be called, but its corresponding clause body altered. For this, code perturbation calls `DeleteSubgoalOfClause/2`, which first locates the correct clause in question, given the irreducible disagreement. Given that the minuend of the last column is zero,

<sup>5</sup> At the time of writing, the input file for the ILP system is crafted manually. For technical reasons, we remove the argument `CurrentColumn` from all clauses; also we added to Progol’s background knowledge the clauses: `sub( X, Y, Z) :- Z is X - Y.` and `leq(X,Y) :- X =< Y.` They also appear in Progol’s `modeb` declarations.

the first clause of `decrement/3` is selected. The perturbation `DeleteSubgoalOfClause/2` now tries repairs to mirror those of the learner. Either the `decrement/3` step is skipped entirely, or some of its subgoals are deleted.

The error `stops-borrow-at-zero` can be reproduced with an entire “no-op”. The following steps are carried out to modify the body of the respective `decrement/3` clause: (i) the input argument of the clause is mapped to its output argument; (ii) all clauses rendered superfluous by this perturbation are removed (see discussion of `borrow-across-zero`); and (iii) the existing first clause of `decrement/3` is replaced by the new perturbed version, which is:

```
decrement(CurrentColumn, Sum, Sum ) :- irreducible,
    last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
    M == 0, !.
```

The modified program reproduces the observed learner behaviour.

**Error: Borrow-across-Zero.** The error `borrow-across-zero` follows the main line of perturbation than the previous error. But rather than repairing the impasse with a “no-op”, an output argument different from the input argument is constructed. For this, one or more steps that are involved in its construction are deleted. Similar to the last error example, a dependency network of variables is created and exploited, as we now explain.

In the given clause, the output argument is `NewSum`; it has an occurrence in `append/3`, where it depends on the variables `NewRestSum` as well as `MM1`, `S`, and `R`, which are all input variables, given the mode declaration of `append/3`. To modify `NewSum`, we consider all subgoals that feature the aforementioned input variables as output variables. Consider `MM1`, which is constructed by the subgoal `MM1 is MM - 1`. When we delete this subgoal, and replace the occurrence of `MM1` by `MM` in `append/3`, we obtain a program that reproduces the error `don't-decrement-zero`, see [11, page 226].<sup>6</sup> Another call to `DeleteSubgoalOfClause/2` is required to obtain `borrow-across-zero`. Now, as the variable `MM` depends on the value of `M`, we can replace the occurrence of `MM` in `append/3` by `M` to yield the intended effect.

#### 4.4 Summary

In all examples, algorithmic debugging correctly indicated the program clause that required manipulation (this is not always the case, see future work on multiple models). Moreover, the cause of the disagreement as well as prior learner performances correctly informed the perturbation actions. Note that our perturbations `DeleteCallToClause/2` and `DeleteSubgoalOfClause/2` mirror the repair strategies that learners perform when encountering an impasse; they often omit or only partially execute steps relevant to the impasse. Our approach proved to be a general, effective, and also computationally inexpensive method. In this

<sup>6</sup> When we only delete the recursive call in the body of `decrement/3`, we obtain `borrow-from-zero`, the fifth most frequent learner error.

paper, we have illustrated the method being able to reproduce the top five subtraction errors of the Southbay study, which together account for 45% of all errors in this study.

## 5 Related Work

There is only little research in the intelligent tutoring systems community that builds upon logic programming and meta-level reasoning techniques. In [1], Beller & Hoppe use a fail-safe meta-interpreter to identify student error. A Prolog program, modelling the expert knowledge for doing subtraction, is executed by instantiating its output parameter with the student answer. While standard Prolog interpretation would fail, a fail-safe meta-interpreter can recover from execution failure, and can also return an execution trace. Beller & Hoppe then formulate *error patterns* which they then match against the execution trace; each successful match is indicating a plausible student bug.

In Looi’s “Prolog Intelligent Tutoring System” [6], Prolog programs written by students are debugged with the help of the automatic derivation of mode specifications, dataflow and type analysis, and heuristic code matching between expert and student code. Looi also makes use of algorithmic debugging techniques borrowed from Shapiro [10] to test student code with regard to termination, correctness and completeness. The Oracle is mechanised by running expert code that most likely corresponds to given learner code, and simple code perturbations are carried out to correct erroneous program parts.

Most closely related to our research is the work of Kawai et al. [3]. Expert knowledge is represented as a set of Prolog clauses, and Shapiro’s Model Inference System (MIS) [10], following an inductive logic programming (ILP) approach, is used to synthesize learners’ (potentially erroneous) procedure from expert knowledge and student answers. Once the procedure to fully capture learner behaviour is constructed, Shapiro’s Program Diagnosis System (PDS), based upon *standard* algorithmic debugging, is used to identify students’ misconceptions, that is, the bugs in the MIS-constructed Prolog program.

While Kawai et al. use similar logic programming techniques, there are substantial differences to our approach. By turning Shapiro’s algorithm on its head, we are able to identify the first learner error with no effort – for this, Kawai et al. require an erroneous procedure, which they need to construct first using ILP.<sup>7</sup> To analyse a learner’s second error, we overcome the deviation between expert behaviour and learner behaviour by introducing the learner’s error into the expert program. This “correction” is performed by small perturbations within existing code, or by the insertion of elements of little code size. We believe these changes to the expert program to be less complex than the synthesis of entire erroneous procedures by inductive methods alone.

---

<sup>7</sup> Moreover, in our approach, given the mechanization of the Oracle, no questions need to be answered by the learner.

## 6 Future Work and Conclusion

In our approach, an irreducible disagreement corresponds to an erroneous aspect of a learner answer not covered by the expert program (or a perturbed version thereof). We have seen that most program transformations are clause or goal deletions, mirroring the repair strategies of learners when encountering an impasse. We have also seen an example where program perturbation meant complementing existing program clauses with more specialized instances. We demonstrated the successful use of Progol to perform a post-processing generalization step to capture learner performance in a general manner.

Our perturbation algorithm could profit from program splicing [12]. Once algorithmic debugging has located an irreducible disagreement between expert performance and learner performance, the respective clause could serve as a *slicing criterion* for the computation of a subprogram (the *program slice*), whose execution may have an effect on the values of the clause's arguments. The program slice could support or replace the variable dependency network we use.

At the time of writing, the input files to Progol are manually written. In the future, we aim at obtaining a better understanding of Progol's inductive mechanism and its tweaking parameter to generate the input file automatically, and to better integrate Progol into the program perturbation process. In the light of [3], we are also considering a re-implementation of Shapiro's MIS and follow-up work to better integrate inductive reasoning into our approach. This includes investigating the generation of system-learner interactions to obtain additional positive or negative examples for the induction process. These interactions must not distract students from achieving their learning goals; ideally, they can be integrated into a tutorial dialogue that is beneficial to student learning.

In practise, the analysis of learner performances across a set of test exercises may sometimes yield multiple diagnoses. Also, some learners may show an inconsistent behaviour (which we have idealised in this paper). In the future, we would like to investigate the automatic generation of follow-up exercises that can be used to disambiguate between several diagnoses, and to address observed learner inconsistencies.

Expert models with representational requirements or algorithmic structure different to the subtraction algorithm given in Fig. 1 might prove beneficial for errors not discussed here. At the time of writing, we have implemented four different subtraction methods: the Austrian method, its trade-first variant, the decomposition method, and an algorithm that performs subtraction from left to right. We have also created variants to these four methods (with differences in sequencing subgoals, or the structure of clause arguments). When a learner follows a subtraction method other than the decomposition method, we are likely to give a wrong diagnosis of learner action whenever the decomposition method is the only point of reference to expert behaviour. To improve the quality of diagnosis, it is necessary to compare learner performance to *multiple* models. We have developed a method to identify the algorithm the learner is most likely following. The method extends algorithmic debugging by counting the number of *agreements* before and after the first irreducible disagreement; also, the "code

size” that is being agreed upon is taken into account (for details, see [13]). In the future, we will combine the choice of expert model (selecting the model closest to observed behaviour) with the approach presented in this paper.

Once an erroneous procedure has been constructed by an iterative application of algorithmic debugging and automatic code perturbation, it will be necessary to use the procedure as well as its construction history to inform the generation of remedial feedback. Reconsider our example Fig. 2(a), where the learner always subtracted the smaller from the larger digit. Its construction history shows that the following transformations to the correct subtraction procedure were necessary to reproduce the learners’ erroneous procedure: (i) deletion of the goal `add_ten_to_minuend/3`, (ii) deletion of the goal `decrement/3`, (iii) shadowing of the goal `take_difference/4` with more specialized instances, and (iv) a generalization step. Some reasoning about the four elements of the construction history is necessary to translate the perturbations into a compound diagnosis to generate effective remediation. For this, we would like to better link the types of disagreements obtained from algorithmic debugging and the class of our perturbations with the kinds of impasses and repair strategies of VanLehn’s theory [11].

There is good evidence that a sound methodology of cognitive diagnosis in intelligent tutoring can be realized in the framework of declarative programming languages such as Prolog. In this paper, we reported on our work to combine an innovative variant of algorithmic debugging with program transformation to advance cognitive diagnosis in this direction.

**Acknowledgments.** Thanks to the reviewers whose comments helped improve the paper. Our work is funded by the German Research Foundation (ZI 1322/2/1).

## References

1. Beller, S., Hoppe, U.: Deductive error reconstruction and classification in a logic programming framework. In: Brna, P., Ohlsson, S., Pain, H. (eds.) Proc. of the World Conference on Artificial Intelligence in Education, pp. 433–440 (1993)
2. Corbett, A.T., Anderson, J.R., Patterson, E.J.: Problem compilation and tutoring flexibility in the lisp tutor. In: *Intell. Tutoring Systems*, Montreal (1988)
3. Kawai, K., Mizoguchi, R., Kakusho, O., Toyoda, J.: A framework for ICAI systems based on inductive inference and logic programming. *New Generation Computing* 5, 115–129 (1987)
4. Koedinger, K.R., Anderson, J.R., Hadley, W.H., Mark, M.A.: Intelligent tutoring goes to school in the big city. *Journal of Artificial Intelligence in Education* 8(1), 30–43 (1997)
5. Maier, P.H.: *Der Nussknacker. Schülerbuch 3. Schuljahr*. Klett Verlag (2010)
6. Looi, C.-K.: Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System. *Instructional Science* 20, 215–263 (1991)
7. Mitrović, A.: Experiences in implementing constraint-based modeling in SQL-tutor. In: Goettl, B.P., Half, H.M., Redfield, C.L., Shute, V.J. (eds.) *ITS 1998. LNCS*, vol. 1452, pp. 414–423. Springer, Heidelberg (1998)
8. Muggleton, S.: Inverse Entailment and Prolog. *New Generation Computing Journal* (13), 245–286 (1995)

9. Ohlsson, S.: Constraint-based student modeling. *Journal of Artificial Intelligence in Education* 3(4), 429–447 (1992)
10. Shapiro, E.Y.: *Algorithmic Program Debugging*. ACM Distinguished Dissertations. MIT Press (1983); Thesis (Ph.D.) – Yale University (1982)
11. VanLehn, K.: *Mind Bugs: the origin of procedural misconceptions*. MIT Press (1990)
12. Weiser, M.: Program Slicing. *IEEE Trans. Software Eng.* 10(4), 352–357 (1984)
13. Zinn, C.: Identifying the closest match between program and user behaviour (unpublished manuscript), <http://www.inf.uni-konstanz.de/~zinn>
14. Zinn, C.: Algorithmic debugging to support cognitive diagnosis in tutoring systems. In: Bach, J., Edelkamp, S. (eds.) *KI 2011*. LNCS, vol. 7006, pp. 357–368. Springer, Heidelberg (2011)