Erschienen in: KI 2014 : Advances in Artificial Intelligence ; 37th Annual German Conference on AI, Stuttgart, Germany, September 22-26, 2014, Proceedings / Lutz, Carsten et al. (Hrsg.). - Cham : Springer, 2014. - (Lecture Notes in Computer Science ; 8736). - S. 37-48. - ISSN 0302-9743. - eISSN 1611-3349. - ISBN 978-3-319-11205-3

Algorithmic Debugging and Literate Programming to Generate Feedback in Intelligent Tutoring Systems

Claus Zinn

Department of Computer Science, University of Konstanz
claus.zinn@uni-konstanz.de

Abstract. Algorithmic debugging is an effective diagnosis method in intelligent tutoring systems (ITSs). Given an encoding of expert problem-solving as a logic program, it compares the program's behaviour during incremental execution with observed learner behaviour. Any deviation captures a learner error in terms of a program location. The feedback engine of the ITS can then take the program clause in question to generate help for learners to correct their error. With the error information limited to a program location, however, the feedback engine can only give remediation in terms of what's wrong with the current problem solving step. With no access to the overall hierarchical context of a student action, it is hard to dose scaffolding help, to explain why and how a step needs to be performed, to summarize a learner's performance so far, or to prepare the learner for the problem solving still ahead. This is a pity because such scaffolding helps learning. To address this issue, we extend the meta-interpretation technique and complement it with a program annotation approach. The expert program is enriched with terms that explain the logic behind the program, very much like comments explaining code blocks. The meta-interpreter is extended to collect all annotation in the program's execution path, and to keep a record of the relevant parts of the program's proof tree. We obtain a framework that defines sophisticated tutorial interaction in terms of Prolog-based task definition, execution, and monitoring.

1 Introduction

The core part of an intelligent tutoring system can be based upon logic programming techniques. The expert knowledge that learners need to acquire is represented as a Prolog program, and the meta-interpretation technique *algorithmic debugging* is used to diagnose learners' problem solving steps. Algorithmic debugging meta-interprets the expert program in an incremental manner. At any relevant stage, it compares its behaviour with the behaviour of the student by making use of a mechanised Oracle. Any deviation between observed learner behaviour to Prolog-encoded expert behaviour is captured in terms of a program location. The deviation can be used by the feedback component of the ITS to address learners' incorrect or incomplete answers. Given the program

clause in question, the feedback engine generates remediation or hints to help learners overcome the error. The feedback engine, however, has no access to the overall hierarchical context of the program clause in question, and also has no information about a learner's past performance and the problem solving steps still ahead. An engine deprived of such information is not capable of generating more sophisticated feedback to further support student learning and motivation. It is also a pity because Prolog's hierarchical encoding of expert knowledge must surely be beneficial to tackling this issue quite naturally. If, in addition, the expert program would be annotated with terms that explain the role of each relevant clause in the overall context, we could further harness the potential of logic programming techniques for computer-aided education. In this paper, we:

- define a simple program annotation language. Each relevant program clause modeling a skill can be associated with a term that describes its use and role in the overall program;
- extend algorithmic debugging to collect all annotations it encounters on the execution path, and to keep a record of all skills tackled; and
- specify a feedback engine that exploits the wealth of information provided by the algorithmic debugger.

A prototype has been implemented to test and show-case our innovative approach to authoring intelligent tutoring systems, and which defines tutorial interaction in terms of task definition, execution and monitoring.

2 Background

Shapiro's algorithmic debugging technique defines a systematic manner to identify bugs in programs [6]. It is based upon a dialogue between the programmer (the author of the program) and the debugging system. In the top-down variant, using the logic programming paradigm, 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 the result of a goal G, then algorithmic debugging passes to the next goal on the goal stack. Otherwise, the goal G is inspected further. Eventually an *irreducible disagreement* will be encountered, hence locating the program's clause where the buggy behaviour is originating from.

Algorithmic debugging can be used for tutoring [7] when Shapiro's algorithm is turned on its head: the expert program takes the role of the buggy program, and the student takes on the role of the programmer. Now, any irreducible disagreement between program behaviour and given answer indicates a *student's* potential error.

We give an example. For this, consider multi-column subtraction as domain of instruction. Fig. 1 depicts the entire cognitive model for multi-column subtraction using the decomposition method. The Prolog code represents a subtraction problem as a list of column terms `(M, S, R)` consisting of a minuend `M`, a subtrahend `S`, and a result cell `R`. The main predicate `subtract/2` determines the

subtract($PartialSum, Sum$) ←
  length($PartialSum, LSum$),
  mc_subtract($LSum, PartialSum, Sum$).

mc_subtract($\_, [], []$).
mc_subtract($CurCol, Sum, NewSum$) ←
  process_column($CurCol, Sum, Sum1$),
  shift_left($Sum1, Sum2, ProcessedColumn$),
  $CurCol1$ **is** $CurCol - 1$,
  mc_subtract($CurCol1, Sum2, SumFinal$),
  append($SumFinal, [ProcessedColumn], NewSum$).

process_column($CurCol, Sum, NewSum$) ←
  last($Sum, LastColumn$), **allbutlast**($Sum, RestSum$),
  subtrahend($LastColumn, Sub$), **minuend**($LastColumn, Min$),
  $Sub > Min$,
  $CurCol1$ **is** $CurCol - 1$,
  decrement($CurCol1, RestSum, NewRestSum$),
  add_ten_to_minuend($CurCol, LastColumn, LastColumn1$),
  take_difference($CurCol, LastColumn1, LastColumn2$),
  append($NewRestSum, [LastColumn2], NewSum$).

process_column($CurCol, Sum, NewSum$) ←
  last($Sum, LastColumn$), **allbutlast**($Sum, RestSum$),
  subtrahend($LastColumn, Sub$), **minuend**($LastColumn, Min$),
  $Sub =< Min$,
  take_difference($CurCol, LastColumn, LastColumn1$),
  append($RestSum, [LastColumn1], NewSum$).

shift_left($\_CurCol, SumList, RestSumList, Item$) ←
  allbutlast($SumList, RestSumList$), **last**($SumList, Item$).

add_ten_to_minuend($CurCol, (M, S, R), (NM, S, R)$) ←
  irreducible, NM **is** $M + 10$.

decrement($CurCol, Sum, NewSum$) ← **irreducible**,
  last($Sum, (M, S, R)$), **allbutlast**($Sum, RestSum$),
  $M == 0$,
  $CurCol1$ **is** $CurCol - 1$,
  decrement($CurCol1, RestSum, NewRestSum$),
  NM **is** $M + 10$, $NM1$ **is** $NM - 1$,
  append($NewRestSum, [(NM1, S, R)], NewSum$).

decrement($CurCol, Sum, NewSum$) ← **irreducible**,
  last($Sum, (M, S, R)$),
  allbutlast($Sum, RestSum$),
  $\not=(M == 0)$
  NM **is** $M - 1$,
  append($RestSum, [(NM, S, R)$], $NewSum$).

take_difference($CurCol$, $(M, S, \_R), (M, S, R)$) ← **irreducible**,
  R **is** $M - S$.

minuend($(M, \_S, \_R), M$).
subtrahend($(\_M, S, \_R), S$).

irreducible.

Fig. 1. The Decomposition Method for Subtraction

number of columns and passes its arguments to `mc_subtract/3`.[1] This predicate processes columns from right to left until all columns have been processed and the recursion terminates. The predicate `process_column/3` receives a partial sum, and processes its right-most column (extracted by `last/2`). There are two cases. Either the column's subtrahend is larger than its minuend, when a borrowing operation is required, or the subtrahend is not larger than the minuend, in which case we can subtract the former from the latter (calling `take_difference/3`). In the first case, we add ten to the minuend (`add_ten_to_minuend/3`) by borrowing from the left (calling `decrement/3`). The decrement operation also has two clauses, with the second clause being the easier case. Here, the minuend of the column left to the current column is not zero, so we simply reduce the minuend by one. If it is zero, we need to borrow again, so `decrement/3` is called recursively. When we return from recursion, we add ten to the minuend, and then reduce it by one.

Consider three solutions to the task of solving the subtraction '32 − 17': Fig. 2(a) depicts the correct solution, Fig. 2(b) an instance of the no-payback error, and Fig. 2(c) a situation with no problem solving steps.

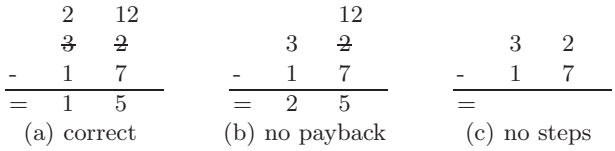

Fig. 2. A correct and two incorrect answers

For the solution shown in Fig. 2(b), our variant of algorithmic debugging generates this dialogue between expert system and the learner:

```
> algo_debug(subtract([(3,1,S1),(2,7,S2)], [(3,1,2),(12,7,5)],ID).

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

ID = (decrement(1,[(3,1,R1),(2,1,R1)]),missing)
```

The dialogue starts with the program's top clause, where a disagreement is found, follows the hierarchical structure of the program, until it ends at a leaf node with an irreducible disagreement that locates the learner's "no payback" error in the decrement skill.

[1] The argument `CurCol` is passed onto most other predicates; it is used to help automating the Oracle and to support the generation of feedback.

With the mechanisation of the Oracle [7], it is not necessary for learners to respond to Prolog queries (or more readable variants thereof). All answers to Oracle questions can be derived from a learner's (potentially partial) answer to a subtraction problem.

Once a program clause has been identified as irreducible disagreement, our previous system used a template-based mechanism for verbalisation, mapping *e.g.*, the disagreement `decrement(1,[(3,1,R1),(2,1,R1)]),missing)` to the natural language feedback "you need to decrement the minuend in the tens" [7].

Note however, that the learner's partial answer in Fig. 2(c) has algorithmic debugging to return the same irreducible disagreement, also by traversing the same intermediate nodes of the proof tree. Consequently, we also generated the same feedback, which left us with a feeling of unease and pedagogical inadequacy. A learner giving the solution in Fig. 2(b), a full albeit incorrect answer, should get a different response than the learner in Fig. 2(c) who provided none of the steps. In general, we believe that the analysis of learner input could profit from embedding the irreducible disagreement in the overall task context. Hence, we will need to enhance our algorithmic debugger to collect *all* relevant information from the program's proof tree. Moreover, we will annotate the expert model with additional information about the logic behind each relevant program clause.

3 Program Annotation Language

Comments are a construct that allow programmers to annotate the source code of a computer program. Comments are used, *e.g.*, to outline the logic behind the code rather than the code itself, to explain the programmer's intent, or to summarize code into quick-to-grasp natural language. In general, comments are ignored by compilers and interpreters, and this also holds for Prolog.

We will use comments in the spirit of the literate programming idea [4]. Our comments construct is a regular Prolog predicate with no effect on the program's semantics. Each comment goal is bound to succeed, and has access to variable bindings in its vicinity:

@**comment**(*Str*, *Arg*) ← (**format**(*Str*, *Arg*), !) ; **true**.

The predicate `@comment/2` is thus mostly syntactic sugar for Prolog's built-in `format/2` predicate. It has two arguments: a string with format specifications, and a list of associated arguments.

Fig. 3 shows a fragment of the expert model for multi-column subtraction, now enriched with comments. Consider the first clause `process_column/3`, which has two comments. Both comments make use of the Prolog goal `get_label/2` that converts the number denoting the current column to a natural language label such as "ones" or "tens" and writes it to the `current_output` stream. The string is then inserted in the result string of `@comment/2`.

The feedback engine assumes that `@comment/2` constructs within the same clause are ordered from low to high specificity. The first occurrence of `@comment/2` in `process_column/3` gives away less information than its second occurrence.

With the annotation, we obtain this execution trace of the expert model:

```
subtract(PartialSum, Sum) ← length(PartialSum, LSum),
   @comment('Subtract␣the␣two␣numbers␣with␣the␣decomposition␣method.', []),
   @comment('Columns␣are␣processed␣from␣left␣to␣right.', []),
   mc_subtract(LSum, PartialSum, Sum).

mc_subtract(_, [], []) ← @comment('Problem␣solved.~n', []).
mc_subtract(CurCol, Sum, NewSum) ←
   @comment('Now,␣process␣the␣~@␣column.~n', get_label(CurCol)),
   process_column(CurCol, Sum, Sum1), CurCol1 is CurCol − 1,
   shift_left(CurCol1, Sum1, Sum2, ProcessedColumn),
   mc_subtract(CurCol1, Sum2, SumFinal),
   append(SumFinal, [ProcessedColumn], NewSum).

process_column(CurCol, Sum, NewSum) ← last(Sum, LastColumn), [...],
   subtrahend(LastColumn, Sub), [...], Sub > Min, CurCol1 is CurCol − 1,
   @comment('In␣the␣~@,␣cannot␣take␣away␣~d␣from␣~d.',
            [get_label(CurCol), Sub, Min]),
   @comment('Need␣to␣borrow␣from␣the␣~@␣before␣taking␣differences.~n',
            [get_label(CurCol1)]), [...]
```

Fig. 3. Annotation of the Expert Program (Fragment)

> *Subtract the two numbers with the decomposition method. Columns are processed from right to left. Now, process the units column. In the units, cannot take away 7 from 2. Need to borrow from the tens before taking differences. Reduce the minuend in the tens. Do this by scoring out the 3 and writing a small 2. Add ten to the minuend. For this, put the figure 1 (representing one tens = 10 units) in front of the 2 units to obtain 12 units. Subtract 7 from 12 to yield 5. Put the figure 5 in the units column. Now, process the tens column. In the tens, the minuend is larger or equal to the subtrahend. Subtract 1 from 2 to yield 1. Put the figure 1 in the tens column. Problem solved.*

Clearly, only selected, situation-specific, parts will be required for tutoring.

4 Extension of Meta-interpretation

The algorithmic debugger that meta-interprets the expert program to identify a learner's error needs to be augmented in three ways:

- rather than terminating with the first irreducible disagreement found, it now traverses the entire program;
- it must keep a record of all the goals visited during code walking;
- it must collect all program annotation attached to goals.

Fig. 4 depicts the enhanced meta-interpreter. The predicate `adebug/5` gets five arguments: a goal, and input and output arguments for agreements and disagreements, respectively. Four cases are distinguished. If the goal is a goal structure `(Goal1, Goal2)`, then the goal `Goal1` is processed first. The results of the first recursive call, updated values for agreements `AINTER` and disagreements `DINTER`,

```
adebug((Goal1, Goal2), AIN, AOUT, DIN, DOUT) ←
  adebug(Goal1, AIN, AINTER, DIN, DINTER),
  adebug(Goal2, AINTER, AOUT, DINTER, DOUT).

adebug(Goal, AIN, AOUT, DIN, DOUT) ←
  on_discussion_table_p(Goal), !,
  copy_term(Goal, CopyGoal), call(Goal),
  ask_oracle(Goal, Answer),
  (Answer = yes
  →
   ( get_annotation(Goal, Ann),
     AOUT = [agr(Goal, Ann, DIN)|AIN], DOUT = DIN ))
  ;
   (get_applicable_clause(CopyGoal, Clause, Ann),
    (irreducible_clause(Clause)
    →
     (AOUT = AIN, DOUT = [irrdis(Goal, Ann)|DIN])
    ;
     adebug(Clause, AIN, AOUT, [dis(Goal, Ann)|DIN], DOUT)
    )))

adebug(Goal, A, A, D, D) ←
  system_defined_predicate(Goal), !, call(Goal).

adebug(Goal, AIN, AOUT, DIN, DOUT) ←
  clause(Goal, Clause), adebug(Clause, AIN, AOUT, DIN, DOUT).
```

Fig. 4. Extended Meta-Interpreter

are passed on to the recursive call to process `Goal2`. The other tree cases process atomic goals. If the `Goal` is relevant enough to be discussed with the Oracle (`on_discussion_table_p/1` holds), we check whether expert behaviour and Oracle agree on the results of calling `Goal`. If we obtain an agreement, we update `AOUT` and `DOUT` accordingly. A record is kept on the agreement, its annotation as well as its history, *i.e.*, its path to the root node, where each node on the path has been disagreed upon. If expert program and Oracle disagree, the disagreement must be examined further. We identify an applicable clause that corresponds to `Goal` – a clause is applicable when its body succeeds when evaluated. If the goal belongs to a clause marked `irreducible`, the disagreement is atomic, and we update the variables `AOUT` and `DOUT` accordingly. Otherwise, we continue to explore the execution tree below `Goal` to identify the irreducible disagreement. The last two clauses of `adebug/5` handle the cases where `Goal` is a built-in predicate, and where `Goal` is not on the discussion table. In the latter, the goal's body is investigated.

If we run algorithmic debugging with an expert program, the learner's answer, and the Oracle, we obtain all the program clauses where learner behaviour matches expert behaviour, and all program clauses where there is no such match. Each agreement and each irreducible disagreement is decorated with the execution path and its annotations.

5 Feedback Engine

We illustrate the design rationale of the feedback engine by reconsidering the two erroneous learner's solutions to the task of solving the subtraction '$32 - 17$'.

5.1 Rationale

For the first solution in Fig. 2(b), the new algorithmic debugger returns two irreducible disagreements: a missing payback operation located at `decrement/3`, and an incorrect difference in the tens at `take_difference/3`. Also, there are two irreducible agreements: the learner correctly added ten to the minuend in the ones at `add_ten_to_minuend/3`, and also took the correct difference in this column at `take_difference/3`. For Fig. 2(c), the algorithmic debugger returns four irreducible disagreements: there are missing operations for decrementing the minuend in the tens, adding ten to the minuend in the ones, taking differences in the ones, and taking differences in the tens. Each of the (dis)-agreements is embedded in its hierarchical structure, and is associated with its annotations.

Both solutions share the same first irreducible disagreement, with the same path to the expert model's top clause. Our old approach, where algorithmic debugging terminates with the first irreducible disagreement, generated only a single feedback candidate to both learners:

`decrement/3` *Reduce the minuend in the tens. Do this by scoring out the 3 and writing a small 2.*

Our new approach exploits the hierarchical context of the first irreducible disagreement and we get these candidates for both of Fig. 2(b) and Fig. 2(c):

`decrement/3` *Reduce the minuend in the tens. Do this by scoring out the 3 and writing a small 2.*
`process_column/3` *In the units, cannot take away 7 from 2. Need to borrow from the tens before taking differences.*
`mc_subtract/3` *Now, process the ones column*
`subtract/2` *Subtract the two numbers using the decomposition method. Columns are processed from left to right.*

For Fig. 2(c), it is rather inappropriate to start with the first and most specific feedback candidate. There is no single agreement in the proof tree, and all irreducible disagreements are of type missing. Here, it is rather better to start with the least-specific feedback associated with the top node, and then to proceed downwards, if necessary.

For Fig. 2(b), it is more appropriate to consider more specific feedback early. Rather than starting with the most specific feedback at `decrement/3`, we back-up to its parent node, which has two child nodes that expert and learner behaviour agree upon. We consider such feedback to be more natural as it acknowledges a learner's past achievements. For Fig. 2(b), we obtain these candidates:

`process_column/3` *In the units, cannot take away 7 from 2. Need to borrow from the tens before taking differences.*

`add_ten_to_minuend/3` *Add(ten to the minuend.*

`take_difference/3` *Subtract 7 from 12 to yield 5.*

`decrement/3` *Reduce the minuend in the tens. Do this by scoring out the 3 and writing a small 2.*

Whenever there is a candidate node that is parent to child nodes that have been agreed upon, the annotations attached to all nodes are verbalised together. For each child node, however, only the least specific feedback is given to learners.

5.2 Algorithm

Whenever the learner asks for help, the current problem solving state is read and sent to the algorithmic debugger for analysis. The list of agreements and disagreements returned are then passed onto the feedback engine, see Fig. 5. The engine's task is to process all information and to compute a sequence of candidate nodes and their associated annotations. For this, `feedback_engine/3` will use the first irreducible agreement and its hierarchical embedding `DPath`. If there is a path to a parent node with agreement children, it will use these nodes as candidate nodes; otherwise it will take the path to the top node and use it in reverse order. The engine takes the first element from the candidate list that does not appear in the dialogue history. Once a `@comment/2` is realized, it is added to the dialogue history. When a user repeatedly clicks on help without advancing the problem solving state, a bottom-out hint will be eventually generated that advances the problem solving state. A subsequent run of the algorithmic debugger will hence return other (dis-)agreement structures, until the task at hand is solved.

give_feedback(*As*, *Ds*) ← **feedback_engine**(*As*, *Ds*, *Acts*),
  realize_acts(*Acts*).

feedback_engine(*As*, *Ds*, *Acts*) ←
  get_first_disagreement_path(*Ds*, *DPath*),
  (**path_to_parent_with_agr**(*As*, *DPath*, *ParentPath*, *AgrNodes*)
  → **combine_nodes**(*ParentPath*, *AgrNodes*, *CandidateNodes*)
  ; **reversePath**(*DPath*, *CandidateNodes*)
  ),
  extract_comments(*CandidateNodes*, *Acts*).

realize_acts(*Acts*) ← **member**(*A*, *Acts*),
  dialogue_history(*DH*), ∀ **member**(*A*, *DH*), **realize_act**(*A*).

Fig. 5. Feedback Engine

6 Discussion

There is little recent research in the ITS community that builds upon logic programming. In [2], Beller & Hoppe use a fail-safe meta-interpreter to identify student error. A Prolog program, encoding the cognitive model, 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 formulate error patterns which are then matched against the execution trace, and where each successful match indicates a plausible student bug.

In [7], we have presented our first meta-interpreter approach to analyse learner input in intelligent tutoring systems. In this initial version, algorithmic debugging terminated with the first irreducible disagreement between Prolog-encoded expert and observed learner behaviour. The disagreement was then directly verbalised by the feedback engine to help learners correct their error. All student actions that were in line with the expert model were ignored. In this paper, we address the drawback and keep a record of all agreements and deviations between expert and learner behaviour and also maintain their hierarchical embedding. Also, we now attach comments to each clause of the expert model.

The association of feedback messages to cognitive models is nothing new. In tutoring systems driven by production rules, expert skills are represented as expert rules and anticipated erroneous behaviour is being simulated by an encoding of buggy rules. Fig. 6 depicts a production rule taken from the the CTAT tutor [1]. It represents one of the skills for adding fractions. The rule's IF-part lists a number of conditions that are checked against the content of a global working memory that captures the current problem solving state. If all conditions are met, the actions in the THEN-part are carried out, usually changing the contents of the working memory. In the THEN part, we find a message construct that is directly used for the generation of remedial feedback. Tutoring systems based on production rules systems perform *model tracing*. After each and every student step, the rule system is executed to identify a rule that can reproduce a learner's action. When such a rule if found, its associated feedback is produced. Each student action is thus being commented on, advancing a tutorial dialogue context by continually tracking student actions. Model-tracing tutors thus keep learners on a tight leash. They have little opportunity to explore different solutions paths; with every step, they are exposed to potentially corrective feedback.

In constraint-based tutors, the correctness of learner input in diagnosed in terms of a *problem state*, given a set of constraints that test whether relevant aspects of the state are satisfied or not. Each relevant but unsatisfied constraint is flagged as potential source of error; its associated feedback message is given to the learner, see Fig. 7. As constraint-based models do not model learner action, there is no hierarchy of skills. Also, developers of constraint-based systems must cope with situations where more than a single relevant constraint is unsatisfied. In ASPIRE's fraction addition tutor, this is addressed by artificially partitioning the problem solving state into discrete units, where the next unit can only be tackled when the current one has been successfully completed. The set of constraints is

```
(defrule one-denominator-multiple-of-other
   (declare (salience 200))
   ?problem <- (problem
      (given-fractions $? ?f1 $?)
      (subgoals))
   ?problem <- (problem (given-fractions $? ?f2 $?))
   (test (not (eq ?f1 ?f2)))
   ?f1 <- (fraction (denominator ?denom1))
   ?f2 <- (fraction (denominator ?denom2)
       (has-converted-form ?conv))
   ?denom1 <- (textArea (value ?d1&:(neq ?d1 nil)))
   ?denom2 <- (textArea (value ?d2&:(neq ?d2 nil)))
   (test (= 0 (mod ?d1 ?d2)))
=>
   (bind ?sub1 (assert (convert-fraction-goal  (fraction ?f2)
      (denominator-value ?d1))))
   (bind ?sub2 (assert (add-fractions-goal (fractions ?f1 ?conv))))
   (modify ?problem (subgoals ?sub1 ?sub2))
   (construct-message
      "[ What denominator could you use to add the two fractions? ]"
      "[ Since " ?d1 " is a multiple of " ?d2 ", use " ?d1 " as the
      denominator of the sum fraction.  ]"
      ))
```

Fig. 6. A Production Rule, see [1]

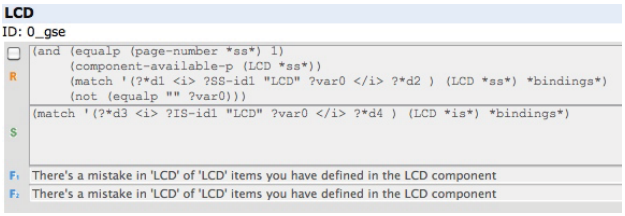

Fig. 7. An ASPIRE constraint, see [5]

de facto divided into subsets that correspond to units. Similar to model tracing, this restricts learners exploring the solution space.

In our approach, learners only get feedback when they explicitly ask for it, and learners may ask for help early or late in their problem solving process. Each help request starts algorithmic debugging anew, now taking into account and acknowledging all learner actions. Hence, we can now create a more natural tutorial dialogue context, which is a huge improvement to our earlier work [7].

7 Conclusion

In this paper, we extend our previous work on using logic programming and meta-level techniques for the analysis of learner input in intelligent tutoring systems. We modified algorithmic debugging to traverse the entire proof tree, and to mark each node as agreeing or disagreeing with learner behaviour. Following the spirit of literate programming, we enrich the expert model with comments that explain the role of each relevant clause in natural language. This is a rather

straightforward idea, and very much in line with the feedback structures attached to production rules or constraints. In combination with algorithmic debugging, the anytime-feedback capability, and the hierarchical representation of expertise, this simple idea is rather powerful, yielding to the generation of sophisticated feedback that is very hard to replicate in the other two approaches.

In the future, we would like to use a fully-fledged natural language generation engine to make generation more flexible (*e.g.*, variation in lexicalisation; tense and formality of language). Also we would like to extend learners' ability to contribute to the dialogue. Instead of pressing the help button, they shall be able to click on any subtraction cell to get help specific to the cell in question. Moreover, we would like to manage multi-turn Socratic dialogues were learners are lead to discover and correct their misconceptions (*e.g.*, [3]). Here, we anticipate the need for additional annotations of the task structure. To further test-drive, validate and fine-tune our approach, we will also implement different domains to explore the use of hierarchically organised annotations and their use to support learning.

References

1. Aleven, V., McLaren, B., Koedinger, K.: Rapid development of computer-based tutors with the cognitive tutor authoring tools (CTAT). In: Proc. of the Int'l Conf. on Artificial Intelligence in Education, p. 990. IOS Press (2005)
2. 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)
3. Chang, K.-E., Lin, M.-L., Chen, S.-W.: Application of the Socratic dialogue on corrective learning of subtraction. Computers & Education 31, 55–68 (1998)
4. Knuth, D.E.: Literate programming. Computer Journal 27(2), 97–111 (1984)
5. Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J., McGuigan, N.: Aspire: An authoring system and deployment environment for constraint-based tutors. Int. J. of Artif. Intell. in Educ. 19(2), 155–188 (2009)
6. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertations. MIT Press (1983), Thesis (Ph.D.) – Yale University (1982)
7. Zinn, C.: Algorithmic debugging to support cognitive diagnosis in tutoring systems. In: Bach, J., Edelkamp, S. (eds.) KI 2011. LNCS (LNAI), vol. 7006, pp. 357–368. Springer, Heidelberg (2011)