Feedback in an interactive equation solver

Harrie Passier

Johan Jeuring

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-021

www.cs.uu.nl

ISSN: 0924-3275

Feedback in an interactive equation solver

Harrie Passier
Open University
Heerlen, the Netherlands

and

Johan Jeuring
Open University
Heerlen, the Netherlands
and
Department of Computer Science
Utrecht University, the Netherlands

Abstract. E-learning tools for mathematical problem solving such as solving linear equations should be interactive. As with pen and paper, a student constructs a solution stepwise. E-learning tools provide the capability to give feedback to a student at each step. Feedback is essential for effective learning and hence crucial for interactive e-learning tools. This paper describes a framework for providing feedback in interactive e-learning tools. The framework is particularly useful for domains with hierarchically structured terms, a set of rewrite rules to rewrite the terms from the domain into other terms, and a well-described goal. The framework is used to give feedback about syntactical errors, about several kinds of semantic errors, and about progression towards a solution. The framework explicitly uses the structure in the data to produce feedback. We discuss an e-learning tool for solving linear equations in which the framework for feedback is used. The techniques for providing feedback are taken from compiler technology and rewriting theory.

1 Introduction

Mathematics is constructive in nature: mathematics students learn to *construct* solutions to mathematical problems. Solving mathematical problems is often done with pen and paper, but e-learning tools offer great possibilities. Interactive e-learning tools that support learning mathematics should provide the capability to give feedback to a student at each step. To illustrate our approach, we will use an e-learning tool for solving a system of linear equations. We call this tool the Equation Solver. Figure 1 shows a screenshot of our tool.

The Equation Solver. The Equation Solver consists of three text fields. The top text field is the working area, in which a student can edit a system of equations stepwise to a solution. The current system of equations is

```
2*x = 3+2*3*x-z-5
y = 3*x-z-5
2*z = 3*x
```

The second text field displays the history of equations. Apparently the previous system of equations was

```
2*x = 3+2*y
y = 3*x-z-5
2*z = 3*x
```

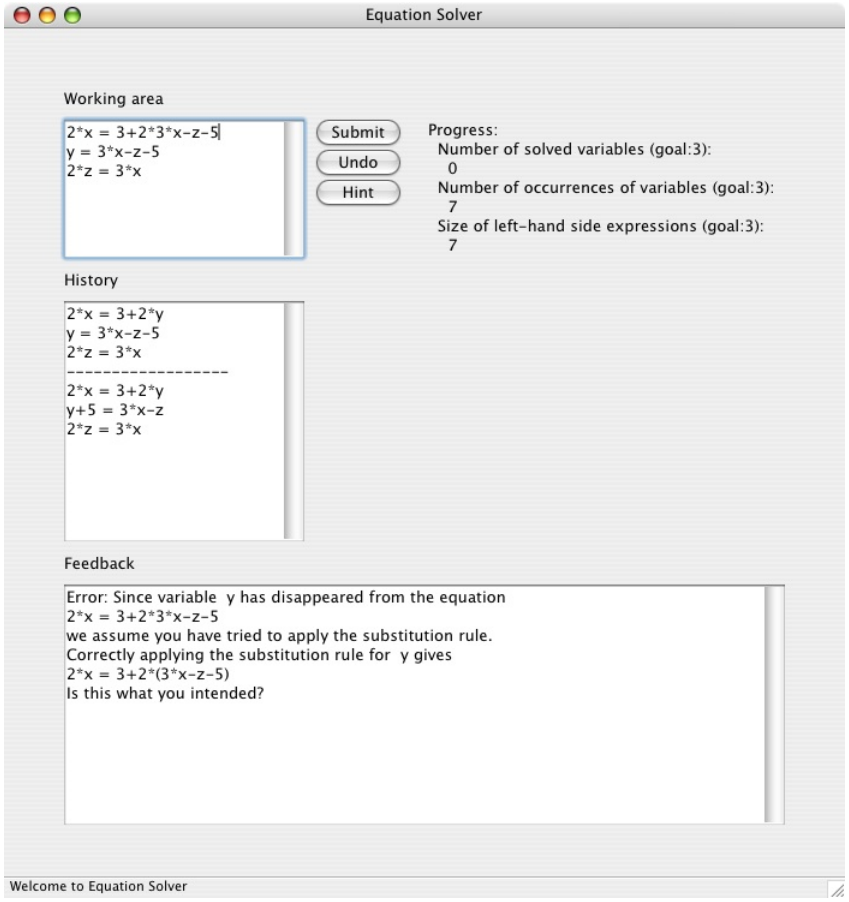

Fig. 1. The Equation Solver

and the student replaced `y` by `3*x-z-5`, forgetting to parenthesize the result. The third text field displays the feedback. In the figure it explains why the last step is incorrect.

The Equation Solver presents a system of equations to a student, for example

```
2*x = 3+2*y
y+5 = 3*x-z
2*z = 3*x
```

and the student has to rewrite these equations into a form with a variable to the left of the equals symbol, and a constant to the right of the equals symbol, for example `x = 7;y = 11/2;z = 21/2`. The student presses the Submit button to submit an edited system of equations, and the Undo button to undo the last step (or any amount of steps). If a student wants help, he or she presses the Hint button to get a suggestion about how to proceed. Finally, the Equation Solver gives information about progress towards a solution by showing how many variables have been solved, and several other kinds of information.

Feedback in the Equation Solver. The Equation Solver gives feedback about two kinds of mistakes:

- syntactical mistakes, for example when a student writes `y+5 = 3*x-` instead of `y+5 = 3*x-z`,
- semantical mistakes, usually mistakes in applying a rewrite rule towards a solution, for example when a student rewrites `y = 3+1` by `y = 5`,

and it gives feedback about (lack of) progression towards a solution.

The Equation Solver consists of a *solver*, which performs symbolic calculations, an *analyser*, which analyses the submitted equations of a student based on a set of rewrite rules for the domain of systems of equations, and several *indicators*, which indicate the progression of a (series of) rewrite step(s).

Note that we try to mimic the pen-and-paper situation as closely as possible, by letting students enter and rewrite equations in a text field. An other approach is to offer the possible rewrite steps to the student, and let the student select a rewrite step, which is then applied to the system of equations by the Equation Solver, as propagated by Beeson [2] in MathXpert. In such a situation, it is impossible to make a syntactical mistake, or to rewrite an equation incorrectly. The former approach has the advantage that a student also learns to enter correct equations, and to choose and apply rewrite steps correctly. Furthermore, it is closer to the pen-and-paper situation. The latter approach has the advantage that a student can concentrate solely on solving a system of equations. The only feedback that needs to be provided in the latter approach is feedback about progression and strategy. Although we do not support the latter approach, it is orthogonal to our approach, and easily added to our tool.

Contributions of this paper. This paper discusses a framework for providing feedback, in which feedback about syntactical mistakes, semantical mistakes, and (lack of) progression in the solving process is produced. The framework assumes a structured domain (like linear equations), for which a set of rewrite rules (or transformations) is defined (like `x+0 = x` for all `x`), a goal is specified (like rewrite all equations to a form where there is a single variable to the left, and a constant to the right of the equality symbol), and one or more measures can be defined with which we can (possibly partly) determine the distance to the goal.

The main results of our work are:

- We show how results from theoretical Computer Science, in particular from the term-rewriting and compiler technology (and in particular parsing) fields, can be used to develop tools that provide semantically rich feedback to students.
- We show how using structural information in data for feedback improves the feedback a tool can give.

We think our framework is useful for several purposes. Developing a tool in our framework forces the developer (a teacher) to be explicit about *all* aspects of a particular domain, and it helps developers of e-learning tools to set up a well-structured feedback component that gives better feedback than existing tools.

In almost all electronic learning environments we know of, feedback is hard coded and/or specified separately for each exercise. Including detailed feedback for exercises is thus very labour intensive. Our framework produces feedback for a whole class of problems. In case of the Equation Solver feedback is automatically generated for all exercises belonging to the class of solving linear equations. Another advantage of our framework is that feedback is produced on the level of rewrite steps the student performs when she solves an exercise, instead of feedback on the final result of the solving process [9]. This is important in cases where different solving methods can be used and solving methods consist of several rewrite steps.

The goal of our research is to develop a general framework to produce feedback. To test our ideas we have implemented a first version of the framework in a tool with which a student can practice solving systems of linear equations. We do not assume the tool replaces a teacher. Rather we assume a teacher first explains the general problem to solve and which solving methods can be used. Then students are able to practice using the Equation Solver and receive semantically rich feedback about errors and progress and, if necessary, hints when they are stuck. The current version of the tool is good enough to test our ideas, but it is not mature enough to be usable in real educational situations. We expect to release a usable version of our tool in the first half of 2006.

To solve a system of linear equations, secondary school students often learn two solving methods. In the first one, called substitution, each variable is expressed in one or more other variables. For example, in the system of equations `x+y = 4; 2*x+y = 9` the variable y in the second equation can be replaced by `4-x` resulting in the system `y = 4-x; 2*x+(4-x) = 9`. After a finite number of such steps the system can be solved. In the second method, one equation is subtracted from another equation. If for example the first equation is subtracted from the second one we obtain the system: `x+y = 4; x = 5`. Again, after a finite number of such steps the system can be solved. A solving method consists of a set of rewrite rules and an order in which the rules are applied to solve the problem.

Of course the methods used in the equation solver should correspond to the methods explained by the teacher, such that feedback and hints correspond to these methods. The framework itself is independent of a solving strategy: solving methods can be added to the Equation Solver. In the present Equation Solver, the rewrite rules of the substitution method are implemented and feedback is given about the correct application of these rules. The order in which a student applies the rewrite rules is not analyzed. This is planned as future work.

Organization. This paper is organized as follows. Section 2 discusses the framework for feedback. Section 3 briefly discusses our approach to syntactic feedback. Section 4 discusses how we provide feedback on individual rewriting steps towards a solution. Section 5 discusses the indicators we have defined and how we help students that seem to be stuck. Section 6 discusses how we can abstract from the Equation Solver to obtain a tool that provides semantically rich feedback for other domains as well. In Section 7 we draw our conclusions, describe related work, and list planned future work.

2 The feedback framework

Our framework for providing feedback assumes we have the following components:

1. A domain with a semantics.
2. A set of rewrite rules for the domain.
3. A goal that can be reached by applying the rewrite rules in a certain order.
4. A set of progression indicators to determine the distance between the goal and the current situation.

Our framework provides feedback about syntactic errors, semantic errors (incorrectly applied rewrite rules), and about progression, using the progression indicators.

To illustrate our framework, we will use the Equation Solver introduced in the Introduction. Solving a system of n linear equations with n variables $x_1, \ldots, x_n$ amounts to finding constants $c_1, \ldots, c_n$ such that $x_1 = c_1, \ldots, x_n = c_n$ is a solution to the system of equations.

We describe the components of our framework for the Equation Solver.

Domain and semantics of the Equation Solver. The domain of the Equation Solver consists of a system of linear equations. We use Haskell [15] types to describe this domain. The top-level type is a list of equations:

```
type Equations      = [Equation]
```

Each equation consists of a left and a right hand expression separated by a '=' (in Haskell denoted by the infix constructor `:=:`) symbol.

```
data Equation = Expr :=: Expr
```

An expression is either zero, a constant, a variable, or two expressions separated by an operator '+', '-', '*', or '/'.

```
data Expr     =  Zero
              |  Con Rational
              |  Var String
              |  Expr :+: Expr
              |  Expr :-: Expr
              |  Expr :*: Expr
              |  Expr :/: Expr
```

The semantics describes how the domain should be interpreted. For the Equation Solver, the semantics is the solution to the system of linear equations.

Rewrite rules for the Equation Solver. A domain has a set of rewrite rules with which terms in the domain can be rewritten.

A rewrite rule rewrites a term of a particular domain to another term of that domain. For example, we have the following rewrite rule for expressions: $(a + b)c \rightarrow ac + bc$, which says that we can rewrite the expression $(a + b)c$ to the expression $ac + bc$ (distribute multiplication over addition) in any context in which this expression appears. For a general introduction to rewrite systems, see Dershowitz et al. [7].

Using rewrite rules, we rewrite terms in the domain to some desired form. For the Equation Solver, the goal is to rewrite the given system of equations to a solution. We now informally present the rewrite rules for the domain of the Equation Solver.

We follow the data representation of the domain and distinguish between rules on the level of a system of linear equations, an equation, and an expression. In these rules a, b, and c are rational numbers, x, y, and z are variables, and e is an expression.

For a system of linear equations we have a single rewrite rule: substitution.

If we have an equation $x = e_1$, we may replace occurrences of x in another equation by e_1. Informally:

$$[x = e_1, \ldots x \ldots = e_2, \ldots] \rightarrow [x = e_1, \ldots e_1 \ldots = e_2, \ldots]$$

For an equation we have four rewrite rules:

$$e_1 = e_2 \rightarrow e_1 \oplus e = e_2 \oplus e$$

where $\oplus$ may be any of $+$, $-$, $*$, or $/$.

For an expression we have a large number of rewrite rules. First, constants may be added, multiplied, etc:

$$a \oplus b \rightarrow c,$$

where c is the rational number sum of a and b if $\oplus$ is $+$, and similarly for $-$, $*$, and $/$. Coefficients of the same variable are summed using the inverse rule of distributing multiplication over addition.

$$a * x + b * x \rightarrow (a + b) * x$$

Furthermore, multiplication (division) distributes over addition (subtraction):

$$(e_1 \oplus e_2) \otimes e_3 \rightarrow e_1 \otimes e_3 \oplus e_2 \otimes e_3,$$

where $\oplus$ may be $+$ or $-$, and $\otimes$ may be $*$ or $/$.

We have only implemented the rewrite rules belonging to the substitution method for solving systems of linear equations at this moment. Implementing the second method, where equations are subtracted from each other, is easy, and will be implemented in the near future.

As argued by Beeson [2], many mathematical operations cannot be expressed by rewrite rules, because they take an arbitrary number of arguments and because other arguments can come in between. Moreover, associativity and commutativity cause problems in rewrite rules. Hence, applying rewrite rules or recognising applications of rewrite rules in user-supplied equations is not a trivial application of pattern matching, but requires more sophisticated programs.

A *normal form* of a term in the domain of a term-rewriting system is a term which cannot be rewritten anymore. The solution of a system of linear equations is a kind of normal form of a system of linear equations, but since we can always add terms to a term, our system does not have normal forms. A term-rewriting system *terminates* if for every term t, we can only rewrite t a finite number of steps. Since we can distribute multiplication over addition and vice versa, our term-rewriting system is clearly not terminating. A term t' is *reachable* from a term t, if there exists a sequence of term-rewriting steps with which we can rewrite t into t'. Clearly, given a solvable system of linear equations, the solution of this system is reachable. In a situation in which terms have normal forms, and the rewriting system is terminating it is much easier to give useful feedback, but for most domains about which we want to give feedback these properties do not hold.

The goal of the Equation Solver. The goal of the Equation Solver is to find constants $c_1, \ldots, c_n$ such that $x_1 = c_1, \ldots x_n = c_n$ is a solution to the system of equations. We assume that all systems of equations set as exercises by the Equation Solver are solvable, a property that is easily verified. The goal is reachable by applying the rewrite rules to the system of equations in a certain order.

Progress indicators for the Equation Solver. To inform a student about the progress in solving a problem, we have defined indicators. An indicator is a measure which (partly) describes the distance from the current system of equations to the solution (the goal). There are several ways to indicate the distance between the currents system of equations and the solution. A possibility is to determine the minimum number of rewriting steps needed to rewrite the current system of equations to the solution. In this paper we investigate indicators that follow the structure of the data. Thus we can provide more specific feedback than just about the distance to the final solution. We have indicators that indicate progress on the level of a system of equations, on the level of a single equation, and on the level of an expression.

In the next sections, we describe how we provide feedback about syntactical errors, semantical errors, and about progression using this framework.

3 Syntax analysis

A student enters an expression in a text field in the Equation Solver. We have to parse this expression in order to analyse it. We use error recovery parser combinators [18] to collect as many errors

as possible (not just the first), and to suggest possible solutions to the errors we encounter. For example, when a student enters `2*x = 3+2*y;y+5 = 3*x-;2*z = 3*x`, the tool reports an error, and says it expects a lower case identifier or an integer in the equation `y+5 = 3*x-`. Furthermore, it proceeds with parsing `2*z = 3*x`, assuming the expression `y+5 = 3*x-<identifier>` has been entered. The parser combinators are very similar to the context-free grammar for the domain of equations. We have tuned the parser such that common errors, such as writing `2y` for `2*y`, are automatically repaired (and reported).

After parsing we perform several syntactic checks, such that the set of variables that occurs in the system of equations hasn't changed, and that all equations are still linear equations, and not, for example, quadratic equations, which would happen if the student would multiply both sides of the equation `2*z = 3*x` by `x`. If such an error occurs, it is reported.

4 Rewriting terms

When a student submits a system of equations to the Equation Solver, the analyser checks if something has changed. If something has changed, the solver checks that the submitted system of equations has the same solution as the original system, and the analyser tries to infer the rewrite rule applied by the student. If the solution of the system has not changed, it is not necessary to determine the rewrite rule that has been applied, but it might still be useful for the student to see the name of the applied rule. If the solution *has* changed, the student has made an error, and it is important to try to report the likely cause of the error.

An important assumption (restriction) we apply here is that we assume a student applies only one rewrite rule per submitted system of equations. In practice, this will not always be the case, but the added complexity of recognizing multiple rewrite rules is left to future work.

In the rest of this section, we discuss the feedback produced by the Equation Solver by means of examples on each of the three levels of our domain. To determine which rule a student intended to apply, we follow a hierarchical approach.

Determining a rewrite on the system of equations level. The analyser starts with trying to find out if the student intended to apply a rule on the level of the system of equations: the substitution rule. The analyser can determine whether or not the substitution rule has been applied by collecting the variables that appear in the different equations. If one variable has disappeared from the set of variables that appear in an equation a substitution step has been applied. Here we assume that an expression such as $x - x$ is internally represented as 0, so that replacing $x - x$ by 0 does not lead to the false conclusion that substitution has been applied. The internal representation is some normal form of the expressions and equations, where occurrences of the same variable are combined. The normalised form of an expression is an expression of the form $a_1 * x_1 + ... + a_n x_n + c$, where each variable occurs once, and all constants have been added in a single constant c. The Equation Solver determines which variable has disappeared, and checks that applying the substitution using that variable leads to the submitted expression. If this is not the case, the Equation Solver reports an error, and shows the correct equation that results from the substitution. For example, if the system of equations `2*x+2*y = 6;y = 4-2*x` is rewritten to `2*x+2*(4+2*x) = 6; y = 4-2*x`, the analyser produces the following error message:

```
Error: Since variable y has disappeared from the equation

                  2*x+2*(4+2*x) = 6

we assume you have tried to apply the substitution rule.
Correctly applying the substitution rule for y results in

                  2*x+2*(4-2*x) = 6

Is this what you meant?
```

There are several things to note about this message: it is only about the equation that contains an error, it tells why it thinks a certain rewrite rule has been applied, and it shows how the correct application of that rule looks.

Determining a rewrite on the equation level. If the analyser has detected a change in the system of equations and in no equation the set of variables that occur has changed, the analyser tries to find out if there exists an equation that has been rewritten. An equation has been rewritten if both the left-hand side and the right-hand side expression of an equation have changed. On the level of an equation, four rewrite rules may be applied: $e_1 = e_2 \rightarrow e_1 \oplus e = e_2 \oplus e$, where $\oplus$ may be any of $+$, $-$, $*$, or $/$. The analyser determines whether or not these rules have been applied by comparing the new equation with the old equation. For example, if the previous system is `2*x+2*y = 5; x-y = 2` and the submitted system is `2*x+2*y = 5; x-y+y = 2+y`, the analyser concludes that the rule: $e_1 = e_2 \rightarrow e_1 + e = e_2 + e$ has been applied on the second equation of the system. In general, the analyser can infer an application of the addition (and subtraction) rule on the level of an equation by calculating the value of the expression `(l-l')-(r-r')`, where `l = r` is the equation in the previous system, and `l' = r'` is the submitted equation. If this value equals 0, then it is likely that the student has performed an addition (or subtraction) step with value `l-l'` on both sides of the equation. If the value equals a constant unequal 0 or a variable (possibly multiplied by a constant), then it is likely that a student has performed an addition step, but has made an error in doing so. This error is reported. Finally, if the value is not a constant or a variable, it is likely that the student has performed a multiplication (or division). To determine if a multiplication step has been performed, the analyser calculates the value of `(l/l') - (r/r')`. If this value equals 0, then it is likely that the student has performed a multiplication (or division) step with value `l/l'` on both sides of the equation. If the value equals a constant, then it is likely that a student has performed a multiplication step, but has made an error in doing so. This error is reported. Finally, if the value is not a constant, something serious is wrong.

Determining a rewrite on the expression level. If no rewrite on the level of a system of equations or on the level of an equation has taken place, the analyser tries to determine if a rewrite on the level of an expression has taken place. It is easy to determine which expression in the system of equations has been changed.

For example, suppose the previous system is `2*(2+y)+2*y = 5; x = 2+y` and the submitted system `2*2+2*y+2*y = 5; x = 2+y`. The analyser infers that the left-hand side expression of the first equation has changed. The analyser checks that the normalised form of the new expression and the previous expression are the same. Furthermore, the analyser tries to infer which expression rewrite rule has been applied. It does this by determining the expression difference between the old expression and the new expression. The expression difference of two expressions consists of the subexpressions that have disappeared from the old expression in the new expression, and the subexpressions that have appeared in the new expression. In the above example, the expression difference is `2*(2+y)` (disappeared) and `2*2+2*y` (appeared). These expressions match the rewrite rule for distributing multiplication over addition. If the normalised form of the new expression and the old expression are different, an error is reported, and the analyser shows all possible correct rewrites of the subexpression that has disappeared from the expression.

The hierarchical approach to determining which rewrite rule has been applied allows us to pinpoint precisely, in many cases, which mistake (likely) has been made.

5 Progression and hints

An indicator gives a distance from the current system of equations to the solution (the goal). It is used to inform a student about progression towards a solution. Before calculating the value of the various indicators, the Equation Solver detects whether or not a student has completed the problem. This is the case if the system of equations is of the form $x_1 = c_1, \ldots, x_n = c_n$. This is easily detected.

We have defined a number of indicators in the Equation Solver.

The first indicator calculates the number of variables for which a student has found a solution. If this number increases the student makes progression.

The second indicator calculates the number of occurrences of variables in a system of equations. Progression is made if this number decreases. For example, in the system `4 + 2*y + 2*x = 5; x = 2 + y` there are four occurrences of variables. Substituting `2+y` for `x` in the first equation reduces the value of this indicator by one. Sometimes the value of this indicator increases due to a substitution, so we do not enforce that the value of this indicator decreases or stays the same at each step.

The third indicator checks if the expression size of the left-hand side expression of an equation has decreased. Since in our solution we want the left-hand side expression of an equation to be a single variable, a reduction in the size of the left-hand side expression (without removing all variables, since in the end a single variable should remain) indicates progression. For example, rewriting the expression `y+3-1` to `y+2` reduces the size of the expression from 5 to 3 (where operators, constants, and variables all count for 1).

The indicators are independent of the rewrite rules in the Equation Solver. So if a student performs a transformation on the system of equations that doesn't change the semantics of the system of equations, but for which the analyser cannot find a corresponding rewrite rule, the indicators can still inform the student about his or her progress.

If a student is stuck, he or she can press the hint button. The Equation Solver will then give a next step, or a hint to help the student to produce a next step. The next step depends on the solving method used. The Equation Solver produces a next step or a hint based on the previous system of equations submitted by the student and the solving method. For example, if the previous system submitted by the student is `4+2*y+2*x = 5; x = 2+y` and the substitution method is used, the system will suggest:

```
Try to substitute 2+y for x in the equation 4+2*y+2*x = 5.
```

We intend to offer various levels of help depending on, for example, the solving method and the tutorial strategy (the strategy for 'tutoring' students in the tool [8]) used. In the above situation, where substitution is used as the solving method, we can think of the following, increasingly detailed, message:

```
Try to apply the substitution rule.

Try to substitute 2+y for x in the equation 4+2*y+2*x = 5.

Substitute 2+y for x in the equation 4+2*y+2*x = 5, resulting in
4 + 2*y + 2*(2+y) = 5
x = 2 + y
```

Different tutorial strategies can be implemented. Besides the messages showed, the system can present for example only the rule that has to be applied and ask the student to apply this rule in the current system of linear equations, or it presents the rule in the context of a simpler task. The last strategy can be valuable in case of solving systems with more than two equations.

6 A general tool

Although the main ideas behind the analysis for feedback in the Equation Solver are reusable in other domains, the implementation is not reusable. For each rewrite rule we have a separate analysis function. All these functions operate on the domain of equations. When a new rewrite rule is added to the system, a new analysis function has to be implemented. When a new domain together with rewrite rules is specified, we have to build a completely new tool. This is labour intensive and requires advanced knowledge and experience. Therefore we want to implement a general tool (a feedback engine [14]), which takes the domain of interest together with rewrite rules as inputs and

automatically transforms the rewrite rules in analysis functions. These functions determine which rewrite rule has been applied if an expression has changed. Furthermore, if an error has been made, the analysis functions determine the closest (using some edit-distance measure) correct solution. The analysis output forms the basis of feedback messages, which can be enriched by tutorial strategies.

The analysis whether or not a rewrite step has been applied and which rewrite rule has been applied, consists of two steps. The first step determines the difference between two expressions of a certain domain. The difference between the expressions is represented by a tuple consisting of terms that are *deleted* and terms that are *inserted* in the expression. The second step tries to explain the *inserted* terms from the *deleted* terms by applying rewrite rules to the *deleted* terms: are the terms in *inserted reachable* from the terms in *deleted* by applying one of the rewrite rules to the terms in *deleted*. Here we assume that only a single rewrite rule has been applied. In case of the Equation Solver, if *deleted* equals `2 + 3` and *inserted* equals `5` than the *inserted* term can be explained by application of the rule $a + b \rightarrow c$ on the *deleted* terms.

If the *inserted* terms cannot be explained by applying a rewrite rule to the *deleted* terms, we try to determine the terms that are closest, under a tree edit-distance measure, to the *inserted* terms after rewriting. We calculate the set of terms obtained by applying a rewrite rule to the *deleted* terms, and then determine the tree edit distance [3] between these terms and the *inserted* terms, and take the one with minimum tree edit distance. The development of a general tool, using generic programming techniques [1], is planned as future work.

7 Discussion

Conclusions. We have introduced a framework for providing feedback in an e-learning tool for a structured domain. Using the structure in the domain, we can provide detailed feedback. The framework consists of a domain with a certain semantics, a set of rewrite rules for the domain, a goal that can be reached by applying the rewrite rules in a certain order, and finally a set of indicators to determine the distance between the desired solution and the current situation. We have used our framework in a prototype e-learning tool for solving a system of linear equations. All of our ideas have been implemented, but we have yet to add the bells and whistles to turn the Equation Solver into a mature tool.

Feedback is crucial in education and is used in many learning paradigms. It is an essential element needed for effective learning. Nevertheless, electronic learning environment courses frequently lack effective feedback [13]. Almost all feedback is related to question-answer situations, is hard coded, and does not use a structural approach. In general, this situation also holds for the field of learning mathematics. If feedback is a crucial element in education and electronic environments increasingly support mathematics education, this gap needs to be filled.

We think our framework will be useful for students, teachers, and e-learning tool developers that build interactive tools in which students have to construct solutions stepwise. It forces a teacher to be explicit about all terms and the semantics of a particular domain, the goal that has to be reached, how progression of the solving process can be measured, and which rewrite rules may be used. It helps the (software) developer to build the feedback component in a structural way. The steps a student can take in such a tool are not limited by a predefined set of rewrite rules provided by the tool, but can be any combination of correct or erroneous steps. The tool tries to recover the rewrite steps taken by the student in order to provide detailed feedback about possible errors. If the tool cannot recover the rewrite rules, the indicators can still help a student in determining whether or not he or she is on the right track. This is important because it is hard, if not impossible, to always determine which sequence of rewrite rules a student has applied.

An advantage of our approach is that feedback is produced for a class of problems. Furthermore, our ideas are reusable for many different classes of problems. In case of the Equation Solver feedback is given for exercises in the class of solving linear equations; instead of dedicated feedback for each separate exercise, a mechanism is defined that produces feedback for all exercises belonging to the class of solving linear equations. We are not aware of similar work. Most of the feedback in

electronic learning environments is hard coded and specified separately for each exercise, for an example see the work by Bokhove et al [4] on providing feedback for exercises.

We have said little about the form and content of the feedback messages. We have shown two messages: one error and one advice message. The error message not only indicates that an error has been made, but also gives the equation that contains the error, and additional information based on the rewrite analysis. This additional information is important because information about the nature of the error and the way it can be corrected is much more effective for learning than simply being informed that an error has been made without any further guidance [11]. This is especially important when students are working in a 'pen-and-paper' like environment, instead of an environment where rewrite rules can be selected from a menu.

Our Equation Solver satisfies most of Beesons [2] eight criteria that must be met if we are to provide successful computer support for education in algebra, trig, and calculus. The first two are *cognitive fidelity*, which means that the software solves the problem in the same way as the student should solve it, and *the glass box principle*, which means that a student can see how the computer solves the problem. To produce feedback and advice about which step to take next, the Equation Solver uses a well-known set of rewrite rules and a solving strategy. The feedback and advice are based on the application of a single rule. Applying multiple rules in a single rewrite step is not supported yet, in the sense that it is allowed, but no feedback is given if an error is made, other than the fact that an error has been made. It follows that our Equation Solver does not completely meet the *customised to the level of the user* criterion. Indicators still help advanced users that apply multiple rules in one step though. The fourth criterion is *the correctness principle*, which means that a student cannot perform incorrect operations. The Equation Solver calculates after each submission the solution of the system of linear equations. If the solution is changed, the analyser will inform the student and, if possible, point out the erroneously applied rule. However, the student can still enter an incorrect equation. We think this has an added advantage: the student becomes aware of the syntax of systems of equations, and learns how to apply rewrite rules. The fifth criterion, *user in control*, means that the student decides what steps should be taken and the computer can help a student when he or she is stuck. As mentioned in the introduction, we try to stay as close as possible to the pen-and-paper situation. Instead of selecting rewrite steps from a menu, a student rewrites equations in a text field. When a student is stuck he or she presses the help button and a next step is produced. This also shows that *the computer can take over if the user is lost* criterion is satisfied. We think our Equation Solver is *easy to use*: no unnecessary typing is required, an infinite Undo is provided, and no unnecessary distractions have been added to the Equation Solver. Finally, the Equation Solver goes beyond the answer-only approach and is thus *usable with a standard curriculum*: it supports a standard curriculum in mathematics, which emphasizes step-by-step solutions.

We have implemented the Equation Solver in Haskell using standard tools from compiler technology, such as parser combinators, and pretty-printing combinators, and using the standard compiler architecture, consisting of a parsing phase, an analysis phase, and a code-generation phase. In our case the code-generation phase is the feedback-generation phase. We intend to make our tool publicly available in the near future. The latest version of the tool can be obtained via email from the authors.

Related work. We have found little literature on structured feedback in e-learning tools, and the way feedback is produced. Most intelligent tutoring systems that have a feedback component use techniques from Artificial Intelligence to report feedback. We think that using the structure in the data and the rewrite rules, we can give more precise and detailed feedback. Of course, there will still be situations where our feedback is insufficient: the amount of possible errors a student can make, and the misconceptions a student can have is close to infinite.

Within the Galois project [4] a digital environment is created in which secondary school students can practice mathematics and perform tests. One of the goals of the project is to automatically provide intelligent feedback. Intelligent feedback is detailed information given to a student, based on an analysis of the students' answer to a question. Feedback should be based on expert knowledge of the mathematical subject, a model of frequently made mistakes, and knowledge

about learning strategies needed to select a suitable feedback form. The authors observe that most of the feedback in electronic environments is primitive and only contains information about the correctness of an answer. They describe how to produce feedback for exercises with a numerical answer. The expected answers are categorized, and provided with feedback. An example of such an exercise is: 'A bicycle tyre has a puncture. Every minute 6 percent of the air escapes from the tyre. Which percentage of air has escaped after 9 minutes?' The expected answers are for example (correct) '43', feedback 'excellent'; '42', feedback 'you are close to the correct answer'; '54'; feedback 'apply rule ...'; ...; 'otherwise' feedback 'incorrect answer'. The expected answers are, besides the correct answer, related to frequently made mistakes and listed by experienced teachers. Feedback is updated automatically if the numbers in such an exercise are changed. An advantage of this approach is the detailed feedback that is given for a certain type of exercise. Disadvantages of the approach are in our opinion the impossibility of reuse for other types of exercises and the lack of feedback on a sequence of solving steps.

Other tools for solving systems of linear equations pay little or no attention to feedback. For example, the Linear System solver (using determinant) [5] returns 'ERROR in perl script on line 23: Illegal division by zero at (eval 129) line 37', if an unsolvable system of equations is entered. Commercial tools such as Algebrator [17] and MathXpert [2] do give feedback on the syntactical level, and hints about making progression, but do not use a structural approach to providing feedback about rewriting steps entered by the student

Cohen et al [6] have developed a tool for solving exercises about computing the derivative of elementary functions. The tool uses rewrite rules called domain rules and decomposes the original problem into sub-problems obtaining a multi-step exercise based on a solution graph. An interactive exercise is then seen as a collection of problems together with the order in which they are solved. According to the students' answer and a predefined strategy, a next step is selected. The correctness of a students' answer is evaluated by a computer algebra system. In this way, a student is guided in solving the initial exercise. Our approach is not based on a solution graph, but uses indicators to inform the student about progression of the solving process and a rewrite analysis to determine which rewrite rule has been (correctly or incorrectly) applied. As a result, the steps a student can take are not limited by a predefined set of rewrite rules or solving strategy, but can be any combination of correct or erroneous rewrite steps. If a step is erroneous, the tool of course complains, but it also tries to give a helpful error message to the student. If the tool cannot recover which rewrite rules have been used, the indicators can still help a student in determining whether or not he or she is on the right track.

ActiveMath [8] contains an approach for representing and processing interactive exercises. The approach relies on a separation of different types of knowledge that have to be handled to generate feedback. An exercise is viewed as a finite state machine of interactions, predefined or generated. Each state represents an interaction of the learner with the system. An interaction contains feedback for the event that triggered it, an interactivity assignment that describes how to substitute certain parts of the feedback by interactive elements, and one or more transition maps describing possible next interactions. The exercise system has the potential to process (partially) generated exercise representations, which depend on the availability of intelligent components that diagnose errors and generate feedback. We think that the rewrite analysis of the Equation Solver is such an intelligent component. The authors observe that finding and encoding all relevant potential erroneous actions of a learner is a bottleneck. As an example, experiments have shown that just for the fraction domain, there exist at least 300 buggy rules. Melis [12] uses the domain of fractions in the design of erroneous examples for ActiveMath. Frequently made errors are listed in terms of misconceptions and buggy rules. The application of rewrite rules plays a central role. We want to determine to what extent these misconceptions can be automatically analysed using our approach.

Heck et al [9] describe a system for diagnostic testing of mathematics students. The system is based on Maple T.A., for automated assessments, and Maple, for verification of the students answers. Classes of exercises can be defined. In the given examples, feedback is just the correct answer together with a short description of how the problem can be solved.

Finally, Marvrikis et al [10] describe a web based learning environment for studying mathematics. The system contains a feedback mechanism that follows an incremental hinting process that changes according to, for example, the time passed, the current goal, and the amount of help a student requests. A mechanism tracks the goals that the author of the activity sets and a student has to reach. The goals involve, for example, selecting an answer for a multiple choice question, putting objects into certain positions, and giving numerical answers. The feedback mechanism comprises production rules defined by an author. It is not completely clear to us how feedback is generated and to which extent the mechanism is reusable.

Future work. The work reported on in this paper is part of our work about feedback [14]. In the future we want to work on several things:

- We want to investigate if we can recognise the application of more than one rewrite rule when comparing a submitted system of equations with the previous system. This probably requires some advanced reachability analysis.
- We want to apply our framework in more domains taken from mathematics.
- We want to allow for nice presentations in our tool, so that for example `(3/2)*x` can be displayed as $\frac{3x}{2}$. The Proxima editor [16], or Amaya [19], might be useful here.
- We want to do more research on types of feedback and the way these can be incorporated in the framework. For example, feedback about the goal structure leads to better performance than feedback about the reasons for the error [11]. Goal related feedback allows students to correct the incorrect actions more often than other types of feedback. Our framework explicitly incorporates the goal and the indicators relate the current system of equations with the goal.
- We want to perform experiments with our tool, in order to obtain experience with different levels of detail in feedback and progression indicators.
- We want to apply our framework to a rather different domain such as UML diagrams: can we apply the framework to provide feedback in an e-learning tool that supports the construction of UML diagrams?

Applying our framework to other mathematical domains, and to more or less structured domains outside mathematics will help us in evaluating the applicability of our ideas.

On a different note: we would like to analyse the behaviour of a student using the Equation Solver to generate new problems, in which rules a student has not applied yet have to be used, or in which the student has to practice rules in applying which he or she made errors solving the last problem.

Acknowledgements. Bert Zwaneveld gave feedback on drafts of this paper. Evert van de Vrie pointed us to the right information. Doaitse Swierstra and Atze Dijkstra helped us in using the Utrecht parsing, scanning, and attribute grammar tools.

References

1. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.
2. M. Beeson. Design principles of Mathpert: Software to support education in algebra and calculus. In N. Kajler, editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer-Verlag, 1998.
3. P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337:217–239, 2005.
4. C Bokhove, A. Heck, and G. Koolstra. Intelligent feedback to digital assessments and exercises (in Dutch). *Euclides*, 2, 2005.
5. Igor Chudov. Linear system solver (using determinant), 2004. See `http://www.algebra.com/algebra/homework/coordinate/linear.solver`.
6. Arjeh Cohen, Hans Cuypers, Dorina Jibetean, and Mark Spanbroek. Interactive learning and mathematical calculus. In *Mathematical Knowledge Management*, 2005.

7. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 243–320, Cambridge, MA, USA, 1990. MIT Press.
8. G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education (ICCE 2005)*, 2005.
9. A. Heck and L. van Gastel. Diagnostic testing with Maple T.A. *Electronic Library of Mathematics of the European Mathematical Society (http://www.emis.de/proceedings)*, 2006.
10. M. Marvrikis, A. Macioncia, and J. Lee. Wallis: a web-based ILE for science and engineering students studying mathematics. *Electronic Library of Mathematics of the European Mathematical Society (http://www.emis.de/proceedings)*, 2006.
11. J. McKendree. Effective feedback content for tutoring complex skills. *Human computer interaction*, 5:381–413, 1990.
12. E. Melis. Design of erroneous examples for ActiveMath. In *Artificial Intelligence in Education (AIED 2005)*, 2005.
13. E. Mory. Feedback research revisited. In D.H. Jonassen, editor, *Handbook of research for educational communications and technology*, 2003.
14. Harrie Passier and Johan Jeuring. Ontology based feedback generation in design-oriented e-learning systems. In P. Isaias, P. Kommers, and M. McPherson, editors, *Proceedings of the IADIS International conference, e-Society*, volume II, pages 992–996, 2004.
15. Simon Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report.* Cambridge University Press, 2003. A special issue of the Journal of Functional Programming, see also `http://www.haskell.org/`.
16. Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents.* PhD thesis, Utrecht University, The Netherlands, Oct 2004.
17. Sofmath. Algebrator – algebra help software, 2005. See `http://www.algebra-help.com/g2-solve-x.html`.
18. S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, volume 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag, 1996.
19. World Wide Web Consortium. Amaya web editor/browser. `http://www.w3.org/Amaya`, 2004.