Feedback Services for Exercise Assistants

Alex Gerdes

Bastiaan Heeren

Johan Jeuring

Sylvia Stuurman

Technical Report UU-CS-2008-018
July 2008

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Feedback Services for Exercise Assistants

Alex Gerdes[1], Bastiaan Heeren[1], Johan Jeuring[1,2], and Sylvia Stuurman[1]

[1] *School of Computer Science, Open Universiteit Nederland, Heerlen, The Netherlands*
[2] *Department of Information and Computing Sciences, Universiteit Utrecht, The Netherlands*

alex.gerdes@ou.nl
bastiaan.heeren@ou.nl
johanj@cs.uu.nl
sylvia.stuurman@ou.nl

Keywords: feedback, web service, exercise assistant, strategy

Abstract

Immediate feedback has a positive effect on the performance of a student practising a procedural skill in exercises. Giving feedback to a number of students is labour-intensive for a teacher. To alleviate this, many electronic exercise assistants have been developed. However, many of the exercise assistants have some limitations in the feedback they offer.

We have a feedback engine that gives semantically rich feedback for several domains (like logic, linear algebra, arithmetic), and that can be relatively easy extended with new domains. Our feedback engine needs to have knowledge about the domain, how to reason with that knowledge (i.e. a set of rules), and a specified strategy. We offer the following types of feedback: correct/incorrect statements, distance to the solution, rule-based feedback, buggy rules, and strategy feedback.

We offer the feedback functionality in the form of light-weight *web services*. These services are offered using different protocols, for example by JSON-RPC. The framework around the services is set up in such a way that it can easily be extended with other protocols, such as SOAP. The services we provide are used in exercise assistants such as MathDox, ActiveMath, and our own exercise assistant.

Our feedback services offer a wide variety of feedback functionality, and therefore exercise assistants using our services can construct different kinds of feedback. For instance, one possibility is to start giving correct/incorrect feedback, and only start to give semantically rich feedback on individual steps when a student structurally fails to give a correct answer. Another possibility is to force the student to take one step at a time, or to follow one specific strategy.

In this paper, we describe the feedback services we offer. We briefly discuss the feedback engine that serves as a back-end to our feedback services. We will give examples of how to use our services. In particular, we will show a web-based application that uses the feedback services in the domain of simple arithmetic expressions.

1 Introduction

Giving feedback is essential for a student's learning process. In a classroom setting, feedback is usually given by a teacher. When a student makes an error a teacher gives feedback to a student on how to get back on the right track. Giving feedback to a number of students is labour-intensive for a teacher.

Many electronic exercise assistants have been developed to support the learning of a student. There exist exercise assistants for practising procedural skills for many domains, including logic, algebra, and calculus. These assistants usually offer a rich user interface, and different kinds of feedback. Exercise assistants have a number of advantages: they can support a large number of students at the same time, and they can give *immediate* feedback. Research (Mory 2003) has shown that during the course of an exercise, under certain circumstances, immediate feedback improves the performance of a student. Providing feedback in exercise assistants is of fundamental importance for their acceptance and usability.

The feedback offered by existing exercise assistants, is often limited, or laborious to specify. Furthermore, some exercise assistants are limited to one domain, whereas others are not able to automatically generate exercises. To our knowledge none of the exercise assistants available today can generate feedback on the *strategy* (or procedural) level.

We have a feedback engine that *automatically* derives various types of semantically rich feedback for a number of domains. We make our feedback functionality available to other exercise assistants in the form of web services. Exercise assistants can extend their functionality with the feedback services we offer.

This paper has the following contributions.

- We describe the feedback services we offer in detail.
- We give an example of how an exercise assistant can use our feedback services.

This paper is structured as follows. Section 2 lists the types of feedback that we have identified in current exercise assistants. Section 3 describes the web-services that can be used to access our feedback functionality, and the details of their interfaces. Section 4 illustrates the use of our services with an example application. Section 5 describes related and future work, and concludes.

2 Feedback

In this section we introduce the various types of feedback that are used in typical exercise assistants. We then describe the problems that exercise assistants face with producing such feedback.

Why should exercise assistants give immediate feedback? Research (Hattie & Timperley 2007) has shown that the use of immediate feedback has a positive effect on the performance of a student. In Rules of the Mind (Anderson 1993), Anderson discusses the effectiveness of feedback in intelligent tutoring systems and observed no positive effects in learning with deferred feedback, but observed a decline in learning rate instead.

Ideally, a tool gives detailed feedback on several levels. For example, when a student rewrites $\frac{1}{2} + \frac{1}{3}$ into $\frac{1}{2} + \frac{2}{}$, our feedback engine will signal that there is

a missing denominator. If $\frac{1}{2} + \frac{1}{3}$ is rewritten into $\frac{2}{5}$, it will signal that an error has been made when applying the rule of adding fractions: correct application of this definition would give $\frac{5}{6}$. Finally, if the student rewrites $\frac{2}{5} + \frac{3}{5}$ into $\frac{4}{10} + \frac{3}{5}$, it will tell that the fractions already have a common denominator and that the numerators should be added.

The first kind of error is a syntax error, and there exist good error-repairing parsers that suggest corrections to expressions with syntax errors. The second kind of error is a rewriting error: the student rewrites an expression using a non-existing or buggy rule. There exist some interesting techniques (Bouwers 2007) for finding the most likely error when a student incorrectly rewrites an expression. The third kind of error is an error on the level of the procedural skill or strategy for solving this kind of exercises.

Existing exercise assistants, such as MathDox (Cohen, Cuypers, Reinaldo Barreiro & Sterk 2003), Aplusix (Chaachoua et al 2004), the Autotool project (Rahn & Waldmann 2002), the Dutch WisWeb (Freudenthal Institute 2004), ActiveMath (Goguadze, González Palomo & Melis 2005), including the slopert project (Zinn 2006), Mathematik Heute (Hennecke, Hoos, Kreutzkamp, Winter & Wolpers 2002), and MathPert (Beeson 1998) are all specialised in a particular type of feedback. We construct a list of feedback types based on the feedback types we found in these exercise assistants:

Correct/incorrect statements. This type of feedback, offered by most exercise assistants, tells whether or not a submitted answer is correct.

Distance to the solution. Another type of feedback indicates the number of steps to the final solution and whether or not a rewritten term is closer to the solution.

Rule-based feedback. This type of feedback gives feedback on the level of rewrite rules, such as: which rules are applicable, or how to perform the application of a particular rule. It can also be used by the student to find out what went wrong with the application of a rule.

Buggy rules. Feedback can also be provided by analysing common mistakes made by students, and distil so-called 'buggy rules'. In case of an incorrect answer/step, the step can be matched against these buggy rules.

Strategy feedback. Many domains, such as logic, algebra, and calculus, require a student to learn strategies. A strategy, also called a procedure, describes how basic steps may be combined to solve a particular problem. For example, at elementary school students have to learn how to add fractions. A possible strategy for adding fractions is: 'First determine a common denominator, add the numerators, and then simplify the resulting fraction'.

There are only very few tools that give feedback at intermediate steps different from correct/incorrect. Although the correct/incorrect feedback at intermediate steps is valuable, it is unfortunate that the full possibilities of e-learning tools are not used. The main reasons probably are that specifying detailed feedback at intermediate steps for an exercise is very laborious, providing a comprehensive set of possible bugs for a particular domain requires a lot of research (see for example Hennecke's (1999) work), and automatically calculating feedback for a given exercise, strategy, and student input is rather difficult.

2.1 Feedback engine

This subsection describes our feedback engine (Heeren, Jeuring, Leeuwen & Gerdes 2008). It is a software application, written in Haskell, which automatically calculates feedback based on an exercise, the strategy for the exercise, and student input. The feedback engine is domain independent; the calculation of feedback does not rely on any domain specific information. This is one of the main reasons why our feedback engine can easily be extended with a new domain.

To automatically derive feedback our feedback engine needs to have a description of a domain (like logic or arithmetic), a set of domain-specific rules (such as the De Morgan rules for the logic domain), and a strategy (such as: rewrite until the expression is in disjunctive normal form). A rule is specified in the form of a rewrite step. Strategies are specified in an embedded domain-specific language in Haskell. Our approach to specifying strategies for exercises is generic and not limited to a particular exercise domain. We have developed a *strategy language* (Heeren et al. 2008), in which we can specify strategies ('first do this', 'repeat this', etc.). Such a specification is in fact a context-free grammar. The sentences of this grammar are sequences of rewrite steps (applications of rules). We can check whether or not a student follows a strategy by parsing the sequence of transformation steps, and checking that the sequence of transformation steps is a prefix of a correct sentence from the context-free grammar.

Besides providing feedback, our feedback engine can generate exercises for any domain. In the domain description we need to describe the domain to a tool named QuickCheck, an automatic testing tool for Haskell, with which our feedback engine can generate random exercises. In the near future we would like to automate this process, by making use of generic programming techniques. The level of difficulty of a generated exercise can be adjusted.

We have implemented a number of domains in our feedback engine, and we keep extending it by adding more domains. At the time of writing our feedback engine offers the following domains (in italics) and exercises:

- *propositional logic*:
  - bringing a proposition in disjunctive normal form (DNF)
- *linear algebra*:
  - orthogonalising a set of vectors in an inner product space (Gram-Schmidt)
  - solving a system of linear equations
  - solving a linear system using Gaussian elimination
  - solving a linear system using matrices
- *arithmetic (including fractions)*:
  - simplifying an arithmetic expression
- *relation algebra*:
  - bringing an expression in conjunctive normal form

Our feedback engine can be easily extended with additional domains. The feedback engine only needs information that is essential to a new domain, according to Bundy (Bundy 1983). The instantiation of a particular domain is a labour-extensive process. Once the domain, rules and strategies are specified we can calculate feedback. It is no longer necessary to construct feedback by hand, which is an error-prone process. Our feedback services even include feedback on the strategy level, which is to our knowledge unprecedented. So far we described *what* our feedback services provide, the next section carries on and describes *how* it is provided.

3 Feedback services

We offer our feedback functionality in the form of a set of online web services. Web services are easier to maintain and deploy than, for instance, a library. Another important reason why we use web services is *abstraction*. By using web services we abstract away from our own implementation details. It enables users of our services to access our functionality without having to know about the details of the back-end feedback engine. The web service interface serves as a contract between the provider and consumer. We can standardise our interface while retaining the possibility of adapting our feedback engine.

Exercise assistants can use our services and provide the feedback types discussed in the previous section. A user of our services is fully in charge of how to use and present feedback to a student.

There are virtually no restrictions on the usage of our feedback services. An example is to start with giving correct/incorrect feedback, and to give semantically rich feedback on individual steps only when a student repeatedly fails to give a correct answer. Another possibility is to choose to only give feedback after the final submission of a student, showing a trace of the steps to the solution.

We can use the input from students/users, via logs and statistics, to optimise our feedback engine, for instance by distilling common mistakes and extracting buggy rules.

3.1 Web services

Our feedback services can be accessed online, via so-called web services. Web services are software systems designed to support inter-operable machine-to-machine interaction over a network. Currently we support three protocols: JSON-RPC (JavaScript Object Notation - Remote Procedure Call), XML-RPC, and a custom protocol for the MathDox project (Cohen et al. 2003) that resembles XML-RPC. The feedback service framework has a modular architecture and can easily be extended with other protocols, such as SOAP. In this document we use JSON-RPC for the examples. The same examples using the other protocols are available at our web-site: `http://ideas.cs.uu.nl/trac`. The JSON-RPC invocation of our feedback services can be done via a CGI (Common Gateway Interface) binary over HTTP.

The following example is a JSON-RPC invocation of the 'allfirsts' service. It calls the service with a list of parameters, here a singleton list containing a four tuple describing the current state. The 'allfirst' service will be explained in detail in Section 3.2. The example is given to highlight the structure of a request URL.

```
http://ideas.cs.uu.nl/cgi-bin/service.cgi?input=
  { "method" : "allfirsts"
  , "params" : [["Simplifying fractions", "[]", "1/2+1/3", ""]]
  , "id"     : 42 }
```

The URL needs to be escaped from illegal characters (like spaces and curly braces), but for clarity reasons we use the normal representation. The CGI binary has one parameter called 'input'. The example service call gets the following reply:

```
{ "result": [ "Rename", "[]", [ "Simplifying fractions"
                              , "[0,2,2,1,0,1]"
                              , "((3 / 6) + (2 / 6))"
                              , "[];" ] ]
, "error": null
, "id": 42 }
```

This reply reports that the rewrite rule 'rename' is applicable, and shows the resulting expression. In the next section we explain the syntax and semantics of the service call and its result in more detail. We also show how to embed the service calls via JSON-RPC in a web application.

An interesting feature of our protocol is that it is *stateless*. When necessary the state is passed around as an extra parameter. We represent the state as a four tuple containing:

- An *exercise identifier*, an overview of exercise identifiers can be found on our web-site (`"Simplifying fractions"`)
- A *location* parameter that holds the remainder of a strategy. The location parameter is encoded as a list of integers. The encoding is rather simple: the integers in the list only encode which element of the firsts set has to be used. We call this a *prefix* because we are in the middle of a derivation to the solution, which means that we have a prefix of a sentence (`"[0,2,2,1,0,1]"`).
- The current *expression* (`"((3 / 6) + (2 / 6))"`).
- An optional *context* parameter denoting the part of the expression we are working on (`"[];"`).

Thanks to the stateless protocol, the state parameter can be saved and the exercise can be continued at a later point. This offers exercise assistants the opportunity to save a student's work.

3.2 Service specification

In this subsection we describe the semantics of all our services together with their input parameters and the output. With the current set of services we can express all types of feedback given in Section 2. The modularity of our feedback services application makes it easy to extend our services with new kinds of feedback. Future services can be accessed in the same way as the current services presented in this section.

Our services can be reached via the following URL:

```
http://ideas.cs.uu.nl/cgi-bin/service.cgi?input=<JSON_input>
```

The general structure of the JSON input parameter that needs to be supplied in the URL is:

```
{ "method" : <service name>
, "params" : <list of parameters>
, "id"     : <request id> }
```

What follows is a description of all provided services. For some we describe the input and output in detail, for others input/output are omitted, because they are obvious.

generate The *generate* service generates a new exercise. It has two parameters: a string representing the exercise identifier, and an integer value for the level of difficulty of the exercise (ranging from one to ten, where ten is the most difficult):

```
{ "method" : "generate"
, "params" : ["Simplifying fractions", 5]
, "id"     : 42 }
```

The result of a *generate* service call is a new state containing a fresh exercise:

```
{ "result": [ "Simplifying fractions", "[]", "2/3+4/5", "[];" ]
, "error" : null
, "id"    : 42 }
```

In addition to the new state the result value contains an error value, indicating whether or not there were any errors, and the identifier value from the request.

derivation The *derivation* service takes a state parameter as input:

```
{ "method" : "derivation"
, "params" : [["Simplifying fractions", "[]", "1/2+1/3", "[];"]]
, "id"     : 42 }
```

and returns the shortest possible route to the solution in a list of three tuples containing the rule identifier, the location, and the resulting expression:

```
{ "result": [ ("Rename",[],"((3 / 6) + (2 / 6))")
            , ("Add",[],"((3 + 2) / 6)")
            , ("AddConst",[0],"(5 / 6)") ]
, "error" : null
, "id"    : 42 }
```

allfirsts The *allfirsts* service takes a state value as a single input parameter and returns a list of three tuples containing a rule identifier, a location, and the state value after applying the rule. The list represents all valid steps a student can take from this location in the strategy. The head of the list is a rule that is the first element of a shortest path to the solution.

onefirst The *onefirst* service gives the head of the list returned by *allfirsts*. This service is redundant; it is in the interface for convenience.

applicable The service *applicable* takes a location and a state as input parameters and returns a list of all rule identifiers that can be applied to the current expression. This list may include rules that do not appear in the strategy.

apply The service *apply* takes a rule identifier, a location and a state as input and applies the rule to the current expression in a new state.

ready The service *ready* determines whether an expression in the state input parameter is a solution. It returns a boolean value.

stepsremaining Given a state, the service *stepsremaining* returns an integer denoting the number of steps towards the solution.

submit The service *submit* takes as input the current state and the student input in the form of an expression:

```
{ "method" : "submit"
, "params" : [ ["Simplifying fractions", "[]", "(1/2)+(1/3)", ""]
             , "(3/6)+(2/6)" ]
, "id"     : 42 }
```

and returns one of the following results, together with some values depending on this result:

- *SyntaxError.* The submitted term is syntactically incorrect.
- *Buggy.* One or more buggy rules match, a list with their identifiers is returned.
- *NotEquivalent.* The student has made a mistake.
- *Ok.* The submitted expression is equivalent and one or more rules have been applied, following the strategy. A list of identifiers of applied rules and a new state are returned.
- *Detour.* The submitted expression is equivalent but one or more of the applied rules do not correspond to the strategy. A list of rule identifiers and a new state are returned.
- *Unknown.* The submitted expression is equivalent but none of the known rules match.

A reply has the following form:

```
{ "result":
    { "result" : "Ok"
    , "rules"  : [ "Rename" ]
    , "state"  : [ "Simplifying fractions", "[0,2,2,1,0,1]"
                 , 3/6+2/6", "[];" ]
    }
, "error" : null
, "id"    : 42 }
```

All the types of feedback given in Section 2 can be constructed with the set of services introduced in this section. The next section gives an example of how to construct feedback with our services.

4 Example application

Consider the following example from the arithmetic domain: add two fractions ($\frac{1}{2}+\frac{1}{3}$). The set of rules consists of: add ($\frac{a}{c}+\frac{b}{c}=\frac{a+b}{c}$), multiply ($\frac{a}{b}\times\frac{c}{d}=\frac{a\times c}{b\times d}$), rename ($\frac{a}{b}=\frac{a\times c}{b\times c}$), and a buggy rule ($\frac{a}{b}+\frac{c}{d}=\frac{a+c}{b+d}$). A possible strategy to solve this type of exercise is the following: find the least common denominator (LCD) of the fractions, rename the fractions such that LCD is the denominator, and add the fractions by adding the numerators.

We use our feedback services to construct different types of feedback when a student tries to solve this exercise.

Correct/incorrect statements. If a student submits $\frac{5}{6}$ as the final answer, we call the 'ready' service. This service indicates that the student has finished the exercise. If a student submits $\frac{3}{6}+\frac{2}{6}$ as a final answer, the 'ready' service reports that this is not a final answer. By calling the 'onefirst' service we can even tell which step the student has to take next.

Distance to the solution. At the start of an exercise the 'stepsremaining' service indicates the (minimum) number of steps it takes to solve the exercise. An exercise assistant can use this information to display a progress bar. We present another example. Suppose a student rewrites the fractions so that they have a common denominator (for example by applying the rename rule), and instead of adding the numerators then renames the fractions again. Although this is a valid rewrite step, it is a detour. The step does not make the distance to the solution shorter. This information can be obtained in two ways: besides the 'stepsremaining' service, the 'submit' service will result in a 'Detour' output.

Rule-based feedback. Whenever a student gets stuck during an exercise, the feedback services can come to aid. The feedback engine knows where a student is in the strategy, and can calculate the expected rule(s) from the strategy. This information is obtained by calling the 'onefirst' or 'allfirsts' services. An exercise assistant can use this information to give the student a hint, for example "Try to apply the rename rule". If a student is still stuck, an exercise assistant can use the 'apply' service to perform the step for the student, showing what she should have done.

Another example of rule-based feedback is to show a student all rules that can be applied to a (sub)expression. This list can be obtained by calling the 'applicable' service.

Buggy rules. What if a student submits an expression that has been rewritten using a buggy rule? In that case the 'submit' service reports that one or more buggy rules match. An example of a buggy rule is to add the numerators and denominators ($\frac{2}{5}$). An exercise assistant can tell the student what error she has made and give a hint, using the 'onefirst' service, about which step she should have taken.

Strategy feedback. The feedback given in the previous types depends on a strategy. A strategy defines the path to a solution. The 'submit' service provides feedback whenever a student deviates from the path to a solution. For example, if a student submits an expression that does not follow the strategy, there are three possibilities:

- the student has made an error, and the feedback engine gives appropriate feedback.
- a student performed a correct rewrite step that is not in the strategy. This is called a detour, and it is up to an exercise assistant to decide what action to take. It can either force a student to follow the strategy or let her carry on with a presumably longer path to a solution.
- the submitted expression is equivalent but the student has applied one or more unknown rules.

This example shows that all types of feedback can be generated using our services.

4.1 Client example

In this subsection we describe how to embed a service in a web application. Although we focus on a web application, the idea can be used in any platform supporting remote procedure calls.

We use AJAX (Asynchronous JavaScript and XML) to obtain an example web application with response time of a desktop application. AJAX uses JavaScript to call a service using an XML-HTTP request. Both the parameters of a service call and the response of a service are expressed in the JSON format. The result values of a service call are placed in the appropriate locations of the web application. Thanks to AJAX this can be done without a page refresh, resulting in a shorter response time.

Many service calls return a new state value. The state value needs to be stored to keep track of a user's progress. An exercise assistant may determine how a state value is stored. It could for example be done in memory or using a cookie.

The following JavaScript code shows a service call:

```
function genExercise() {
  var exercise = $("exercise");
  var myAjax = new Ajax.Request
    ( "http://ideas.cs.uu.nl/cgi-bin/service.cgi",
        { method     : 'post'
        , parameters : 'input = { "method" : "generate"
                               , "params" : ["Simpifying fractions", 5]
                               , "id"     : 42}'
        , onSuccess  : function(response) {
            var resJSON = response.responseText.parseJSON();
            exercise.innerHTML = resJSON.result; }
        }
    );
}
```

Note that this code uses the well known 'prototype' JavaScript library, which provides a clean interface to create AJAX requests. The *genExercise* function declares an *exercise* variable that points to an HTML element with the identifier 'exercise'. A successful service call updates the HTML element with the result value. The next HTML fragment shows how to display a button that invokes the *genExercise* function and a text field, with identifier 'exercise', which contains the result value after a service call.

```
<html>
  <body>
    <div align="center"><br/><br/>
      <button id="generate">Generate exercise</button><br/><br/>
      <strong>Generated exercise:</strong>
      <pre id="exercise">...</pre>
    </div>
  </body>
</html>
```

The function *genExercise* is then bound to the generate button as follows:

```
window.onload = function() {
  $("generate").onclick = genExercise;
  ...
}
```

This small example shows how semantically rich feedback can be obtained with relatively little effort.

5 Conclusion

This paper has introduced feedback services that can be used by exercise assistants to provide various types of semantically rich feedback. We have described the interface of the feedback services and given examples of how to use and call our services, and how to embed our services in a web application.

We have shown that our feedback services can produce all the feedback types that we have identified.

Our services are already being used by some exercise assistants: MathDox and our own web application. We are working on a connection between our services and the ActiveMath tool.

We have set up a wiki and issue tracking system (Trac) for the development of our feedback software and services. This system also provides access to our software repository, via an online source code browser. It can be reached via: `http://ideas.cs.uu.nl/trac`.

Related work There are many exercise assistants that offer an environment in which students can practice skills by solving exercises. Examples are MathDox, Aplusix, the Autotool project, ActiveMath and MathPert, to mention just a few. They usually offer a rich user interface and immediate feedback to the student. However, most of these assistants limit their feedback to correct/incorrect messages.

The ActiveMath project also offers services (Libbrecht & Winterstein 2005). These services offer functionality for creating complete courses. Our services focus on the, more detailed, exercise level.

Future work We will continue our research and try to improve our feedback engine and services, by adding more domains and protocols to increase the number of exercise assistants that can use our feedback services.

References

Anderson, J. R. (1993), *Rules of the Mind*, Lawrence Erlbaum.

Beeson, M. J. (1998), Design principles of Mathpert: Software to support education in algebra and calculus, *in* N. Kajler, ed., 'Computer-Human Interaction in Symbolic Computation', Springer-Verlag, pp. 89–115.

Bouwers, E. (2007), Improving automated feedback – a generic rule-feedback generator, Master's thesis, Utrecht University.

Bundy, A. (1983), *The Computer Modelling of Mathematical Reasoning*, Academic Press.

Chaachoua et al, H. (2004), Aplusix, a learning environment for algebra, actual use and benefits, *in* 'ICME 10 : 10th International Congress on Mathematical Education'. Retrieved from `http://www.itd.cnr.it/telma/papers.php`, January 2007.

Cohen, A., Cuypers, H., Reinaldo Barreiro, E. & Sterk, H. (2003), Interactive mathematical documents on the web, *in* 'Algebra, Geometry and Software Systems', Springer-Verlag, pp. 289–306.

Freudenthal Institute (2004), 'Wisweb', `http://www.fi.uu.nl/wisweb/en/welcome.html`.

Goguadze, G., González Palomo, A. & Melis, E. (2005), Interactivity of exercises in ActiveMath, *in* 'International Conference on Computers in Education, ICCE 2005'.

Hattie, J. & Timperley, H. (2007), 'The power of feedback', *Review of Educational Research* **77**(1), 81–112.

Heeren, B., Jeuring, J., Leeuwen, A. v. & Gerdes, A. (2008), Specifying strategies for exercises, *in* S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki & F. Wiedijk, eds, 'AISC/Calculemus/MKM 2008', LNAI 5144, Springer-Verlag, pp. 430 – 445.

Hennecke, M. (1999), Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen, PhD thesis, Hildesheim University.

Hennecke, M., Hoos, M., Kreutzkamp, T., Winter, K. & Wolpers, H. (2002), *Mathematik heute: Unterrichtssoftware Bruchrechnung, Diagnostisches Lehr-Lern-System zur Bruchrechnung für das Lehrwerk "Mathematik heute".*, Hannover: Schroedel Verlag.

Libbrecht, P. & Winterstein, S. (2005), The service architecture in the activemath learning environment, *in* N. Capuano, P. Ritrovato & F. Murtagh, eds, 'First International Kaleidoscope Learning Grid SIG Workshop on Distributed e-Learning Environments', British Computer Society.

Mory, E. (2003), Feedback research revisited, *in* D. Jonassen, ed., 'Handbook of research for educational communications and technology'.

Rahn, M. & Waldmann, J. (2002), The leipzig autotool system for grading student homework, *in* M. Hanus, S. Krishnamurthi & S. Thompson, eds, 'Functional and Declarative Programming in Education (FDPE)'.

Zinn, C. (2006), Feedback to student help requests and errors in symbolic differentiation, *in* 'Intelligent Tutoring Systems', Vol. 4053 of *Lecture Notes in Computer Science*, Springer, pp. 349–359.