

Feedback Services for Exercise Assistants

Alex Gerdes¹, Bastiaan Heeren¹, Johan Jeuring^{1,2}, and Sylvia Stuurman¹

¹ *Faculty of Computer Science, Open University, Heerlen, The Netherlands*

² *Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands*

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

Keywords: feedback, service, exercise assistant, strategy, web-based

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 lacunas regarding to the feedback they offer.

We have a framework that gives semantically rich feedback for several domains (like logic, linear algebra, arithmetic), and can be relatively easy extended with new domains. We need 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 a light-weight *web services*. These services are offered using different protocols, for example by JSON-RPC. The framework is set up in such a way that it can be easily extended with other protocols, like SOAP. The services we provide are already used by existing exercise assistants, for instance by MathDox, ACTIVEMATH, and our own exercise assistant.

Our feedback services offer a wide variety of feedback functionality, exercise assistants using our services can construct different kinds of feedback. One possibility for instance, 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, our main focus is to 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 an example web-based application that uses the feedback services, for the domain of simple arithmetic.

1 Introduction

Giving feedback is essential for a student's learning process. In a classroom setting the feedback is given by a teacher. When a student goes astray a teacher gives feedback to a student 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 a teacher in this regard. Exercise assistants support a student to practice skills in many domains, such as logic, algebra, calculus, etc. They usually offer a rich user interface, and different kinds of feedback. Exercise assistants have the ability to support a large number of students. Another advantage of an exercise assistant is to give *immediate* feedback, which is nearly impossible for a teacher to do when the number of students grows. Research (Mory 2003) has shown that during exercises, 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.

There are many exercise assistants (Beeson 1998, Chaachoua et al 2004, Cohen, Cuypers, Reinaldo Barreiro & Sterk 2003, Gogvadze, González Palomo & Melis 2005), but the feedback they offer is usually limited, or laborious to specify. Some exercise assistants are limited to one domain, whereas others are not able to generate exercises and have to be listed by hand. 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 provides various types of feedback, listed in section 2, and is able to *automatically* derive semantically rich feedback for a number of domains. We have made our feedback functionality available to other exercise assistants. Exercise assistants can extend their functionality with the feedback we offer.

The contributions of this paper are:

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

The document is structured as follows. Section 2 lists the types of feedback we provide. We continue to describe the web-services that are to be used to access our feedback functionality in section 3. Section 4 describes the interface details and illustrates the use of our services with an example application. Finally we describe related and future work and conclude in section 5.

2 Feedback

An important question that should be answered in the first place is: why should exercise assistants give immediate feedback? Research (Hattie & Timperley 2007, Morrison, Ross, Gopalakrishnan & Casey 1995) 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.

Erev et al. (Erev, Luria & Erev 2006) also claim that immediate feedback is often to be preferred.

Ideally a tool gives detailed feedback on several levels. For example, when a student rewrites $\frac{1}{2} + \frac{1}{3}$ into $\frac{1}{2} + \frac{2}{3}$ 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 according to the strategy the numerators should be added.

The first kind of error is a syntax error, and there exist good error-repairing parsers (Swierstra & Azero Alcocer 1999) that suggest corrections to formulas 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 already 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.

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 supporting detailed feedback for each exercise is very laborious, providing a comprehensive set of possible bugs for a particular domain requires a lot of research (see for example (Hennecke 1999)), and automatically calculating feedback for a given exercise, strategy, and student input is very difficult.

The following list gives an overview of the different types of feedback:

Correct/incorrect statements. This type of feedback, offered by most exercise assistants, tells whether or not a given solution is correct. An example of an exercise assistant offering this kind of feedback is formed by the Dutch WisWeb (Freudenthal Institute n.d.).

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. From this knowledge a *progress bar* can be constructed, indicating the number of steps a student still has to take in order to solve the exercise. An example of an exercise assistant offering that kind of feedback is APLUSIX.

Rule-based feedback. This type of feedback gives feedback on the level of rewrite rules. Examples of this type of feedback are: which rules are applicable or give a hint about how to perform the application of a particular rule. This kind of feedback can be used by the student to find out what went wrong with the application of the rule.

Another approach is to use a set of valid rewriting rules within a certain domain. A non-valid rewriting, submitted by a student, is compared to this set of valid rewritings; the closest one is then used to ask the student whether he or she maybe meant to use that specific rewriting rule.

Buggy rules. Feedback can also be provided by analysing common mistakes made by students, and distill so-called ‘buggy rules’. In case of an incorrect

solution/step, the step can be matched against these buggy rules.

Feedback that has been constructed using buggy rules can inform the student what mistake was made. Buggy rules are usually labour-intensive to specify. An example of the buggy rules approach can be found in AlgeBrain (Alpert, Singley & Fairweather 1999). This approach can be combined with the rule-based approach, as is done in the SLOPERT project (Zinn 2006)

An example of a buggy rule is to add the numerators and denominators, when adding fractions, e.g. $\frac{1}{2} + \frac{1}{3} = \frac{2}{5}$.

Strategy feedback Many domains, such as logic, algebra, or 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 strategy for adding fractions is: ‘First find a common denominator, add the numerators, and try to simplify the resulting fraction’. If a student does not start with determining a common denominator, feedback on the strategy level might be: ‘Before adding fractions, they should have a common denominator’.

We developed a feedback engine (Jeuring 2007, Heeren, Jeuring, Leeuwen & Gerdes 2008, Heeren & Jeuring 2008) that *automatically* calculates the aforementioned feedback. In the next section we discuss how other exercise assistants can use our feedback functionality.

3 Feedback services

We offer our feedback functionality as online web services. Web services are better maintainable and easier to deploy than, for instance, a library. Exercise assistants can use our services and provide their users the feedback types discussed in the previous section. With the diversity in types of feedback and the simple interface (discussed in section 4) the user of our services is fully in charge how to use and present the feedback to the students.

There are virtually no restrictions in the usage of our feedback services. An example 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 choose to only give feedback after the final submission of a student, showing a trace of 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 extract buggy rules from them.

3.1 Domains

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). Rules are specified as a set of rewrite steps. Strategies are specified in an embedded domain-specific language

in Haskell (Peyton Jones 2003). Our approach to specify 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.). From this specification we can create something that looks like 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 every domain. In the domain description we need to describe the domain to a tool named QuickCheck (Claessen & Hughes 2000), 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 (Backhouse, Jansson, Jeuring & Meertens 1999). The difficulty of an exercise is parameterised.

Our feedback engine is equipped with a number of domains 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*:
 - bring a proposition in disjunctive normal form (DNF),
- *linear algebra*:
 - Gram-Schmidt,
 - solving a linear system,
 - Gaussain elimination,
 - solving a linear system with matrices,
- *arithmetic (including fractions)*:
 - simplifying arithmetic expressions
- *relation algebra*:
 - bring an expression in conjunctive normal form.

Let us consider an example from the arithmetic domain, the problem of adding two fractions, for example, $\frac{1}{2} + \frac{1}{3}$. The set of rules consist 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, add the fractions by adding the numerators.

Let’s have a look at the different types of feedback we may encounter when a student tries to solve the exercise.

Correct/incorrect statements. If the student submits $\frac{5}{6}$ as the final answer, the feedback engine will indicate that the exercise has finished. If the student submits a wrong answer, and no buggy rule can be applied, our

feedback engine indicates that the submission was incorrect. Another possibility is that the following expression is submitted: $\frac{3}{6} + \frac{2}{6}$. Our feedback engine will recognise that the final solution has not been reached and will report that another step can be made. It can even tell which step the student has to take next.

Distance to the solution. At the start of the exercise the feedback engine can indicate the (minimum) number of steps it takes to solve the exercise. An exercise assistant can use this information to display a progress bar. Another way in which this information is useful: a student has rewritten the fractions so that they have a common denominator (i.e. applied the rename rule), instead of taking the next logical step (adding the numerators) the student renames the fractions again. Although this is a valid rewrite step, it isn't the most efficient. The step taken does not make the distance to the solution any smaller. An exercise assistant can convert this information into an appropriate feedback message.

Rule-based feedback. Whenever a student gets stuck during an exercise, the feedback engine can come to aid. The feedback engine knows where a student resides in the strategy and can deduce from the strategy the applicable rule(s). This information can be returned to the student as a hint, e.g. "Try to apply the rename rule". If a student still remains stuck, the feedback engine can apply the rule and take the step for the student showing what she should have done.

Buggy rules. If a student makes an error, the feedback engine checks if a buggy rule (a common mistake) can be matched. If a rule can be matched, this is reported to the exercise assistant. 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 rule-based feedback, which step she should have taken.

Strategy feedback The feedback given in the previous types are already related to a strategy. A strategy defines the route towards the result. If a student deviates from the route, there are two possibilities: the student has made an error, and the feedback engine gives appropriate feedback, or a student performed a correct rewrite step that is not in the strategy. This what we call a detour, it is up to the exercise assistant to decide what action to take, either to force a student to remain within the strategy or let her carry on and let her take a less efficient or unknown route.

Our feedback engine can be easily extended with additional domains. We only need information that is essential to a 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 have describe *what* our feedback services provide, the next subsection carries on and describes *how* it is provided.

3.2 Web services

The feedback services can be accessed online, via so-called web services. Web services (W3C 2004) are a software system designed to support inter-operable machine-to-machine interaction over a network. It has an interface described in a machine-processable format.

A protocol is a set of formal rules describing how to transmit data across a network. Currently we support three protocols: JSON-RPC (JavaScript Object Notation - Remote Procedure Call), XML-RPC and a custom protocol for the MathDox (Cohen et al. 2003) project. However, the feedback service application has a modular architecture and can be easily extended with other protocols, like 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>.

JSON is a data interchange format, it is a lightweight format and easy for humans to read and write. It is easy for machines to parse and generate. JSON is built on two structures: a collection of name/value pairs and an ordered list of values. JSON-RPC can be used to access our feedback services. JSON-RPC is a remote procedure call protocol encoded in JSON. It is a very simple protocol, defining only a handful of data types and commands. The JSON invocation of our feedback services can be done via a CGI (Common Gateway Interface) binary over HTTP. An example with some indentation to show the structure of a request URL, the service itself will be explained in section 4:

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

Note that 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 results in the following result reply:

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

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 (we appended the values from the previous result reply example between parentheses):

- an *exercise identifier*, an overview of exercise identifiers can be found on our web-site ("Simplifying fractions"),
- and 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 first set has to be used. We call this a *prefix* because we are in the middle of a derivation to the end 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 in time. This offers exercises assistants the opportunity to save a student's work.

4 Services specification

In this section we describe the semantics of every service along with its input parameters and the output. The name of the service, listed underneath, is also a hyper-link to the actual service. Our services can be reached by 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"      : <not used yet> }
```

What follows is a description of all provided services. For some we describe the input and output in detail, the remaining ones have a similar structure.

generate The *generate* service, as its name indicates, generates a new exercise. It has two parameters: a string representing the exercise identifier and an integer value for the difficulty of the exercise:

```
{ "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 }
```

Next to the new state the result value entails an error value, indicating if there were any errors, and an unused identifier value.

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 path to the end 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 a new state value if the rule would have been applied. 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 leads to the shortest path to the end solution.

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

applicable The service *applicable* takes a location and a state as input parameter and returns a list of all rule identifiers that are applicable to the current expression. The list includes rules that are not in the strategy.

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

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

stepsremaining The service *stepsremaining* returns an integer denoting the number of steps towards the end solution, given the state.

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

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

and returns one of the following result codes, together with some values depending on the result code:

- *SyntaxError*. A syntax error occurred.
- *Buggy*. One or more buggy rules could be matched, 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 in line with the strategy. A list of applied rule identifiers 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 given back.
- *Unknown*. The submitted expression is equivalent but none of the known rules match.

In JSON the result 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 }
```

Every type of feedback, set out in section 2 can be constructed with the set of services introduced in this section. For example, the ‘distance to the solution’ type of feedback is enabled by the *stepsremaining* service. The ‘buggy rules’ type of feedback is delivered by the *submit* service.

4.1 Client example

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

We use AJAX (Asynchronous JavaScript and XML) to give the example web application the response time of a desktop application. AJAX uses JavaScript to call a service using a XML-HTTP request. The parameters of the service call are in JSON. The response of a service call is in JSON as well, the exact structure of a response value has been explained in the previous subsection. The result values of a service call be placed in appropriate areas 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 in order to keep track of the user’s progress. How the state value is stored, is entirely up to the exercise assistant. It could, for example, be done in memory or in a cookie.

The next piece of 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 make AJAX requests. The *genExercise* function declares a *exercise* variable that points to a 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 will contain 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>

```

This small example shows semantic rich feedback can be obtained with a relatively small effort. The expressions in this HTML example are encoded in plain ASCII. To improve the usability, a better representation can be given by also using non-ASCII characters (like ÷). However, these improvements are domain specific and are not trivial to automate. For instance exercises involving matrices are hard to represent, especially if the dimensions change during an exercise.

5 Conclusion

This paper introduces our feedback services that exercise assistants can use to provide semantically rich feedback. The services include feedback on several levels and in a number of different types. We describe the interface of the feedback services and give examples of how to call our services and how to embed our services in a web application.

Our services are already being used by some exercise assistants: MathDox and our own web application (Lodder, Jeuring & Passier 2006, Passier & Jeuring 2006). 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 by the following URL: <http://ideas.cs.uu.nl/trac>.

Related work There are many exercise assistants that offer students an environment in which they can practise skills by solving exercises, such as MathDox (Cohen et al. 2003), APLUSIX (Chaachoua et al 2004), the Autotool project (Rahn & Waldmann 2002), ACTIVEMATH (Gogvadze et al. 2005) and MathPert (Beeson 1998), to mention just a few. They usually offer a rich user interface and immediate feedback to the student. However, most of them are limited to correct/incorrect feedback, because it is often difficult and laborious in these systems to catch mistakes.

The ACTIVEMATHproject also provide services (Libbrecht & Winterstein 2005). Their services operate on a different level, and offers functionality for creating whole courses. Our services focus on the exercise level.

Future work We 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.

A topic that needs further research is how to let others, for example teachers or domain experts, specify exercises, strategies and feedback messages.

References

- Alpert, S. R., Singley, M. K. & Fairweather, P. G. (1999), ‘Deploying intelligent tutors on the web: An architecture and an example’, *International Journal of Artificial Intelligence in Education* **10**(2), 183–197.
- Anderson, J. R. (1993), *Rules of the Mind*, Lawrence Erlbaum.
- Backhouse, R., Jansson, P., Jeurig, J. & Meertens, L. (1999), Generic programming - An Introduction -, in S. D. Swierstra et al., eds, ‘Advanced Functional Programming’, Vol. 1608 of *LNCS*, Springer-Verlag, pp. 28–115.
- 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.
- Claessen, K. & Hughes, J. (2000), Quickcheck: A lightweight tool for random testing of haskell programs, in ‘Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2000’, pp. 286–279.
- 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.

- Erev, I., Luria, A. & Erev, A. (2006), ‘On the effect of immediate feedback’, <http://telem-pub.openu.ac.il/users/chais/2006/05/pdf/e-chais-erev.pdf>.
- Freudenthal Institute (n.d.), ‘Wisweb’, <http://www.fi.uu.nl/wisweb/en/welcome.html>.
- Gogvadze, 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. (2008), Recognizing strategies, in A. Middeldorp, ed., ‘(to appear in) WRS 2008, Reduction Strategies in Rewriting and Programming, 8th International Workshop’.
- Heeren, B., Jeuring, J., Leeuwen, A. v. & Gerdes, A. (2008), ‘Specifying strategies for exercises’, Submitted for the 9th International Conference on Intelligent Tutoring Systems, ITS-08.
- Hennecke, M. (1999), Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen, PhD thesis, Hildesheim University.
- Jeuring, J. (2007), Feedback in exercise assistants, in ‘Online Educa Berlin, 13th International Conference on Technology Supported Learning & Training’.
- Libbrecht, P. & Winterstein, S. (2005), The service architecture in the active-math 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.
- Lodder, J., Jeuring, J. & Passier, H. (2006), An interactive tool for manipulating logical formulae, in M. Manzano, B. Pérez Lancho & A. Gil, eds, ‘Proceedings of the Second International Congress on Tools for Teaching Logic’.
- Morrison, G. R., Ross, S. M., Gopalakrishnan, M. & Casey, J. (1995), ‘The effects of feedback and incentives on achievement in computer-based instruction’, *Contemporary Educational Psychology* (20), 32 – 50.
- Mory, E. (2003), Feedback research revisited, in D. Jonassen, ed., ‘Handbook of research for educational communications and technology’.
- Passier, H. & Jeuring, J. (2006), Feedback in an interactive equation solver, in M. Seppälä, S. Xambo & O. Caprotti, eds, ‘Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006’, Oy WebALT Inc., pp. 53–68.
- Peyton Jones, S. (2003), *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press.
- 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)’.

- Swierstra, S. D. & Azero Alcocer, P. R. (1999), Fast, error correcting parser combinators: a short tutorial, *in* J. Pavelka, G. Tel & M. Bartosek, eds, ‘SOFSEM’99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics’, Vol. 1725 of *LNCS*, pp. 111–129.
- W3C (2004), ‘Web Services Glossary, W3C Working Group Note 11’, <http://www.w3.org/TR/ws-gloss/>.
- 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.