Feedback services for stepwise exercises

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2014-005
February 2014

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 stepwise exercises

Bastiaan Heeren[a,], Johan Jeuring[a,b]

[a]*School of Computer Science, Open Universiteit Nederland*
[b]*Department of Information and Computing Sciences, Universiteit Utrecht*

Abstract

Advanced learning environments such as intelligent tutoring systems for algebra, logic, programming, physics, etc. let a student practice with stepwise exercises, and support a student solving such exercises by providing feedback. These environments usually provide various types of feedback, for example about the correctness of a step, common errors, hints about how to proceed, or complete worked-out solutions. Calculating feedback is generally delegated to a dedicated expert knowledge module, also known as a domain reasoner. Existing architectural descriptions of learning environments do not precisely specify the interaction between this module and the rest of the learning system. We propose a design based on the stateless client-server architecture that clearly decouples the expert knowledge module from the learning environment. We describe a set of feedback services that support the inner (interactions within an exercise) and outer (over a collection of exercises) loops of a learning system, and that provide meta-information about a class of exercises, such as solving quadratic equations, or performing Gaussian elimination. The feedback services do not depend on a particular domain and are based on the various feedback types described in the literature.

The paper analyzes which domain-specific knowledge about an exercise class is needed for implementing the feedback services. Based on this analysis, we developed a framework for implementing domain reasoners that offers generic functionality such as rewriting, simplifying, and comparing terms. We have implemented several domain reasoners in this framework, both for external learning environments and for simple prototypes. The proposed design is evaluated with these implementations, and we reflect on our experience with developing domain reasoners.

Keywords: intelligent tutoring systems, domain reasoners, feedback services

---

Email addresses: `Bastiaan.Heeren@ou.nl` (Bastiaan Heeren), `J.T.Jeuring@uu.nl` (Johan Jeuring)

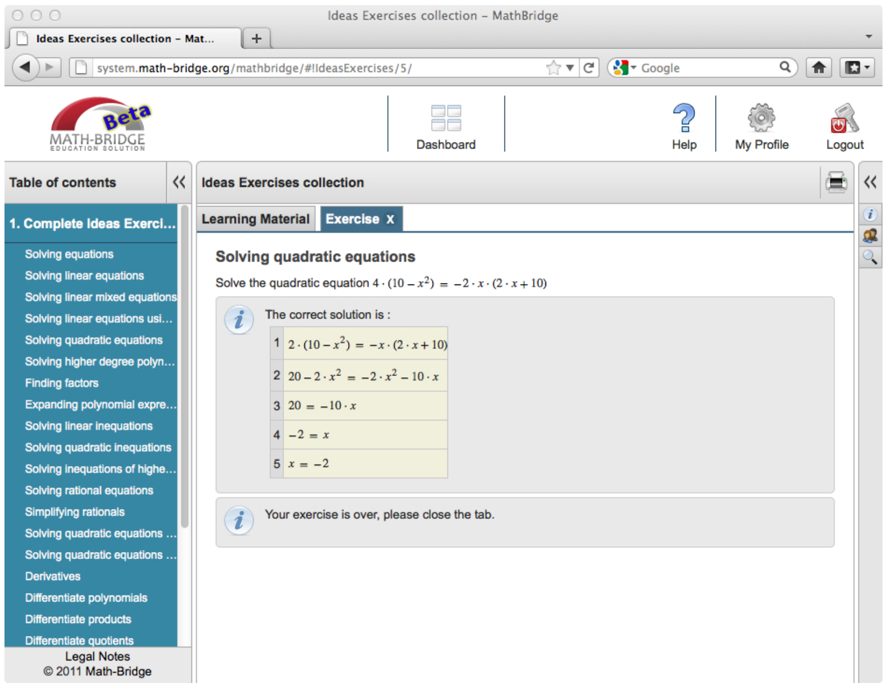
1. Introduction

Innovative learning practices make use of technology to support learning by doing, to simulate real-life situations where learners improve their technical and problem-solving skills, to combine learning and assessment in new ways, to give teachers feedback about progress of their students, and to analyze student learning so that students can steer their learning [16]. Examples of such technology for learning are intelligent tutoring systems [17], adaptive hypermedia [9], serious games [53], etc. Such interactive tools let students practice, analyze student interactions, give feedback on student actions, and help students make progress.

A task in a learning environment can take many different forms: it can be a multiple-choice question, an essay question that is corrected off-line by a teacher or automatically analyzed, a question that asks for an expression from a particular domain (what is/are the solution(s) of $4(10 - x^2) = -2x(2x + 10)$, or give Newton's second law of motion, which relates acceleration, mass, and force), or a question that a student typically solves stepwise. For example, in a learning environment for mathematics that supports solving an exercise stepwise, an exercise about quadratic equations might be solved as in Figure 1. Stepwise exercises are particularly popular in learning environments for mathematics, such as MathDox [14, 15], the Digital Mathematical Environment (DME) of the Freudenthal Institute [19], Math-Bridge [52] (based on ActiveMath [38]), APlusIx [13], the Carnegie Learning Algebra tutor, etc. Environments such as the DME and Math-Bridge offer thousands of stepwise exercises to a student. But stepwise exercises are also used in logic [36], physics [55], programming [23], and many more domains.

Usually, technology for learning distinguishes correct from incorrect answers or interactions, and often such technology provides other kinds of feedback to the learner too. The literature on feedback [41, 50] distinguishes several types of feedback, such as knowledge about correct performance, about how to proceed, about bugs or misconceptions, and approximately ten other types. A number of these types need information about the knowledge and progress of a student, which is usually captured in a student model [7]. A student model can vary between recording which exercises a student has successfully completed to maintaining an ontology and using student interactions as proof that a student masters particular competencies modeled in the ontology.

The sequence of tasks offered by a learning environment is often called the *outer loop* [54]. The outer loop selects a task for a user, probably based on knowledge about the student in the student model. The *inner loop* presents the selected task to a student, and lets the student work on the task. When working on a task, a learning environment can offer various types of feedback to a learner. It can report whether or not a student answer is correct, whether the exercise is solved correctly, what next step a student can take, etc. There are many possibilities here, and different learning environments have made different choices. In this paper we discuss the design of a software architecture for offering feedback to learners when solving a stepwise task in a learning environment.


Ideas Exercises collection – MathBridge
system.math-bridge.org/mathbridge/#!IdeasExercises/5/
Dashboard
Help
My Profile
Logout
Table of contents
1. Complete Ideas Exerci...
Solving equations
Solving linear equations
Solving linear mixed equations
Solving linear equations usi...
Solving quadratic equations
Solving higher degree polyn...
Finding factors
Expanding polynomial expre...
Solving linear inequations
Solving quadratic inequations
Solving inequations of highe...
Solving rational equations
Simplifying rationals
Solving quadratic equations ...
Solving quadratic equations ...
Derivatives
Differentiate polynomials
Differentiate products
Differentiate quotients
Legal Notes
© 2011 Math-Bridge
Ideas Exercises collection
Learning Material
Exercise X

Solving quadratic equations

Solve the quadratic equation $4 \cdot (10 - x^2) = -2 \cdot x \cdot (2 \cdot x + 10)$

The correct solution is :

1	$2 \cdot (10 - x^2) = -x \cdot (2 \cdot x + 10)$
2	$20 - 2 \cdot x^2 = -2 \cdot x^2 - 10 \cdot x$
3	$20 = -10 \cdot x$
4	$-2 = x$
5	$x = -2$

Your exercise is over, please close the tab.

Figure 1: A worked-out solution in Math-Bridge

Traditionally, the architecture of an intelligent tutoring system (ITS) is described by means of four components [44], as depicted in Figure 2.

1. The expert knowledge module
2. The student model module
3. The tutoring module
4. The user interface module

This division into conceptual modules is still visible in recent research on intelligent tutoring systems [43]. From a software architectural viewpoint, the meaning of the arrows in Figure 2 is not very precise. It is clear that the various components need information from each other, but what kind of information, and how this information is communicated, is left informal. This paper focuses on providing expert knowledge whenever a learning environment wants to offer feedback to a learner. We describe webservices for the expert knowledge module that offer facilities for reasoning about the domain studied in the learning environment. The webservices are used by the other components of an intelligent tutoring system to obtain domain knowledge. We view the other components of an intelligent tutoring system as a single entity, which we will refer to as the learning environment. We will precisely define what kind of information is provided by the expert knowledge module, and how this information is communicated to a learning environment. Essentially, a learning environment queries

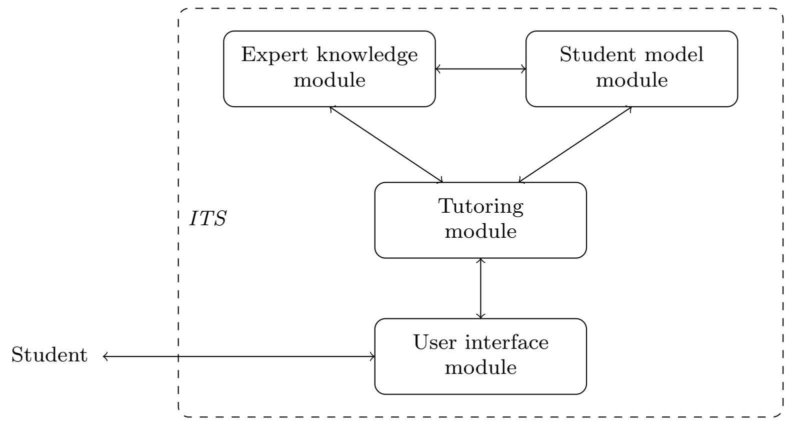


Figure 2: Four component ITS architecture

the expert knowledge module by means of a request, providing the necessary information for the expert knowledge module to calculate a response.

Our webservices offer the types of feedback listed in the feedback literature: is a step submitted by a learner correct, is it an instance of a common mistake, what is a good next step to take, what does a solution to this exercise look like, etc. Our various services are closely linked to feedback types, and offer a more fine-grained approach to providing expert knowledge than existing component-based [47] or agent-based [48] approaches.

Besides the design of a software architecture for the expert knowledge module of an intelligent tutoring system, this paper also discusses an implementation of the architecture, and our experience with using the implementation to offer feedback services to several learning environments. Although we will illustrate our ideas by means of software for calculating feedback to be used in intelligent tutoring systems, we claim that our webservices for feedback are also useful for serious games and other learning environments. We will discuss design decisions for developing components which offer functionality for analyzing and giving feedback on student interactions. The design decisions are based on software design principles and on our experience with developing and deploying such components for a number of learning environments.

This paper has the following goals:

- We discuss the design of an intelligent tutoring system with a clearly separated expert knowledge module that offers webservices for calculating feedback. The design is based on the stateless client-server architecture.
- We relate the webservices to known feedback types, and categorize the services into three groups: inner loop, outer loop, and meta-information.
- We analyze what domain-specific knowledge is needed for implementing a tutor that offers stepwise problem solving support, and explain how a

generic framework can provide the webservices based on this knowledge.

- We report our experience with developing several domain reasoners and interacting with other learning systems.

This paper is organized as follows. Section 2 introduces the concept of a domain reasoner, a component that calculates feedback for a particular domain. We will argue why it is important to calculate feedback using domain reasoners, and compare domain reasoners with approaches used by current learning environments. Section 3 discusses what kind of feedback services for stepwise exercises are required by learning environments. We will then review the feedback types that are found in literature (Section 4) and relate these types to the feedback services. Section 5 presents rewriting strategies, the central concept we use for calculating feedback, and the other components of an exercise class that are used for implementing the feedback services. Section 6 evaluates the proposed design and discusses the non-functional quality attributes that are relevant for providing feedback services. Section 7 presents the lessons we learned in using our feedback services in various learning environments. Section 8 discusses related work and Section 9 concludes.

2. Domain reasoners

An intelligent tutoring system consists of many components. The most important components are (1) an expert knowledge module, which includes a collection of exercises, (2) a student model to track the progress of a student and to adapt the system to the level of the student, (3) a tutoring component which based on a student model, the learning goals of a student, and the expert knowledge module, selects material for a student to study and practice, and (4) an interface in which a student can study the material and work on the tasks [44]. Many tutoring systems also have a monitoring module for teachers, and an authoring environment to author new content or exercises.

The intelligent tutoring literature usually distinguishes a separate component for an expert system that can 'reason about the problems' [17], 'an expert knowledge module' [44], or 'a domain expert' [51]. Following Goguadze [25], we will use the term *domain reasoner* for this component. The development of domain reasoners supporting stepwise solving of tasks is the focus of this paper.

A domain reasoner knows about the objects in a domain, how objects can be manipulated, and how to guide the manipulation of objects to reach a particular goal. For a stepwise exercise, a domain reasoner can construct a solution to a task, a hint on how to proceed with a next step, or recognize that a student has made a common error (applied a 'buggy rule'). A domain reasoner only returns domain knowledge, such as a worked-out solution, a hint, or a buggy rule. It does not determine how this information is presented to a student. For example, suppose a student has to solve the equation $x^2-4x+3=0$ and asks for a hint. The learning environment asks a domain reasoner for solving quadratic equations to calculate a hint, given the starting expression $x^2-4x+3=0$. The domain reasoner returns the *rewrite rule* `nice-factors`, together with the pair $(-3, -1)$ and the result of applying the rule. The rule `nice-factors` is used to factorize a quadratic expression, in this case into $(x-3)(x-1)$. It is up to the learning environment to present this information in a suitable manner to the student. By returning domain knowledge instead of textual feedback messages, a learning environment can also use the information returned by the domain reasoner to update a student model, or to collect information to report to a teacher.

2.1. Design considerations

When developing a domain reasoner component, we have to answer several fundamental questions:

- Is the domain reasoner an external, separate component reusable by other learning environments, or is it embedded in a learning environment?
- Is the domain reasoner developed with the goal of giving feedback in mind, or is it a component with a more general purpose, such as a simulator, or a computer algebra system?
- Is the feedback offered by the domain reasoner specified in an individual exercise ('in the following exercise, a common error is to ...') or is it computed for a class of exercises such as the class of quadratic equations?
- Is the approach to calculating feedback tied to a particular domain, such as calculating the partial derivative of a function, or is it generic, and used for various kinds of stepwise exercises?

We briefly discuss each of the above questions.

External or embedded. Although the intelligent tutoring literature usually distinguishes a separate component for domain reasoning, quite a few intelligent tutoring systems have an embedded domain reasoner. Such a system can fine-tune a domain reasoner to its own purposes. Embedding a domain reasoner in an intelligent tutoring system makes it hard for external environments to use the embedded domain reasoner, let alone adapt it. In addition, this approach has the risk that the domain reasoner is too specialized to the particular tutoring system to reuse it in another intelligent tutoring system. Indeed, we have only found a single tutoring system [27] of which the domain reasoning component is reused by another tutoring system [25]. In contrast to most of the other components of an intelligent tutor, such as the expert knowledge module, the tutoring component, and the user interface, a domain reasoner for different tutors provides similar, if not the same, functionality. How to solve a quadratic equation, which rules may be applied, correctness of a step, etc., are all the same in the various tutors for mathematics.

Feedback-oriented or general purpose. For some domains, domain reasoners are readily available. For example, many learning environments for mathematics use a computer algebra system (CAS) for the component that analyzes the actions of a student and provides feedback to a student [25, 49]. A CAS often contains a lot of domain-specific knowledge about evaluating mathematical expressions, and can, for example, check that a step taken by a student does not change the final solution of an exercise, and thus answer the question about whether or not a step is correct, and whether or not an exercise is correctly solved. Even for simple exercises, checking an answer for correctness with a CAS is much harder than it looks due to subtle consequences of the built-in equality operator of the CAS [6]. It is even harder, if not impossible, to use a CAS for determining a next intermediate step, showing the steps by means of which a solution is calculated, or signaling that a student has made a common error. A CAS has no knowledge about how a student solves a problem, and often uses algorithms for solving mathematical problems that are very different from the solving procedures a student should use.

Individual exercise or exercise class. Sometimes it is cumbersome to use a domain reasoner to give feedback to students, or there is no domain reasoner available. The DME [19], MathDox [14, 15], and Math-Bridge [52] offer the possibility to hardcode custom feedback in an exercise. For a typical multi-step exercise, this sometimes increases the size of the exercise measured in lines of text by a large factor. For example, the specification of an exercise for adding two fractions extended with feedback about correctness, application of a buggy rule, and omitted simplification steps takes 106 lines in MathDox [18]. Hardcoding feedback in an exercise in which a student practices applying the different components of a standard procedure is infeasible, because the number of exercises in such learning environments is huge, and because hardcoding feedback in an exercise makes it very hard to consistently adapt feedback to a user, or to change feedback. Math-Bridge contains over a thousand exercises in which a student practices solving linear equations, quadratic equations, quadratic inequations, higher-order equations, etc., and solving these exercises follows a standard procedure. Hardcoding feedback in each of these exercises is infeasible and undesirable.

Particular domain or generic. Most domain reasoners have been developed for a particular domain. For example, Zinn's domain reasoner for symbolic differentiation [56] has been developed specifically for the purpose of giving feedback in stepwise exercises about symbolic differentiation. It is not easy to reuse components of this domain reasoner to calculate feedback for, for example, exercises about multi-column subtraction [57]. ACT-R [2] is a theory with a set of principles for developing cognitive tutors. The ACT-R theory is embodied in a software environment developed on top of Common Lisp. The environment provides some reusable components for specifying goals etc., but still requires a substantial amount of programming for developing a domain reasoner. For most domains, developing a domain reasoner is a challenging task and requires

learning environments	
– DME [19]	web-based learning environment by the Freudenthal Institute for secondary math education
– Math-Bridge [52]	e-learning platform for online bridging courses in mathematics, the successor of ActiveMath [38]
– MathDox [14, 15]	software tools for creating interactive mathematical documents by Eindhoven University of Technology
prototypes	
– Logic tool [36, 35]	bringing propositions into disjunctive normal form, and proving equivalences between logical formulae
– Ask-Elle [23]	stepwise development of simple functional programs

Table 1: Tools with a connection to our domain reasoners

a serious investment. Already in 1995, Anderson et al. [2] noticed that the technical accomplishment [in developing a domain reasoner] is 'no mean feat'.

2.2. Proposed design

We propose to develop a domain reasoner for stepwise exercises as a separate component, which can be called by learning environments, but need not be part of such an environment. Thus different learning environments can share the same domain reasoner, and not every learning environment developer needs to develop a separate domain reasoner. We think a domain reasoner should be developed on top of a generic framework for specifying rewrite steps and *strategies* for solving exercises, so that many aspects related to domain reasoners can be inherited from the generic framework and development costs can be reduced. Domain reasoners reside on a server, and learning environments call *feedback services* [22] delivered by the domain reasoners. This is a standard stateless client-server architecture for delivering feedback services.

Our design clearly decouples the domain reasoner component from the rest of the learning environment and enables a more precise description of the functionality of a domain reasoner. Decoupling the domain reasoner from the rest of the system might raise questions about the interaction between the domain reasoner and components such as the student model and the authoring environment. We will address this issue in Section 7.1.

The design we propose is based on building domain reasoners for real tutoring systems: see Table 1 for an overview. Several externally developed learning systems now use our domain reasoners. Furthermore, a number of research prototypes have been developed that have been used in classroom settings. The implemented domain reasoners span a variety of different topics and emphasize the genericity of our approach.

Figure 3 illustrates the proposed architecture for feedback services. The learning environments using feedback services appear to the left of the dashed line. The learning environments, the clients, call feedback services on the server

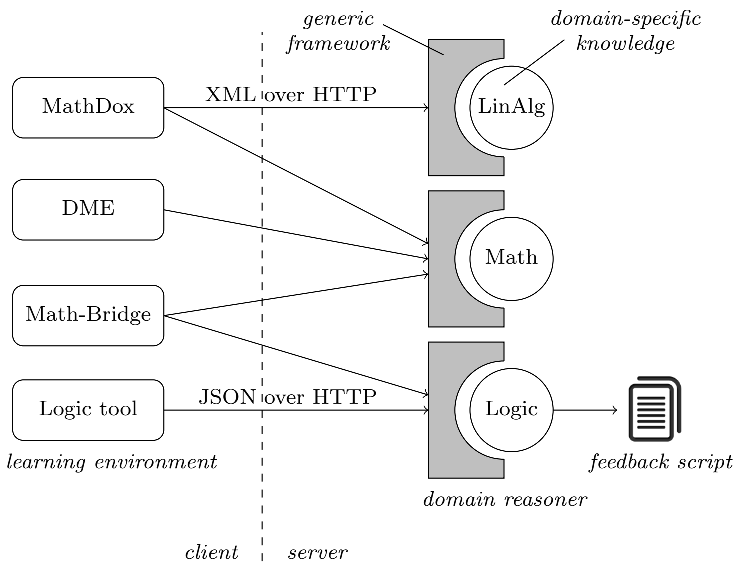


Figure 3: Learning environments using feedback services provided by domain reasoners

via JSON or XML over HTTP. The feedback services are provided on the server by domain reasoners, which appear to the right of the dashed line. A domain reasoner uses domain-specific information, shown as a circle, such as rewriting rules for logic, and a strategy that describes how to solve an exercise in which a logic expression has to be rewritten to disjunctive normal form. Furthermore, it uses a shared generic framework for techniques such as rewriting, simplifying, comparing, searching, etc., which is shown as the grey area surrounding the circles. Some domain reasoners offer configurable feedback scripts to translate abstract domain knowledge to textual feedback messages, to support learning environments that do not want to translate abstract domain knowledge to textual feedback messages themselves.

Note that multiple learning environments use the same domain reasoner, and that a single learning environment can call multiple domain reasoners. A learning environment can call services from other tools too, such as a CAS. For example, Math-Bridge uses at least four different external tools for providing feedback to students. To integrate the results from different external tools, Goguadze has developed a query language for feedback services for interactive exercises [24].

3. Feedback services for tutoring systems

What kind of services for stepwise exercises should a domain reasoner deliver? At the outer-loop level of a tutoring system, a domain reasoner can help

outer loop	
– examples	predefined example exercises of a certain difficulty
– generate	makes a new exercise of a specified difficulty
inner loop	
– allfirsts	all possible next steps (based on the strategy)
– apply	application of a rewrite rule to a selected term
– diagnose	analyze a student step (details in Figure 4)
– finished	checks whether response is accepted as an answer
– onefirst	one possible next step (based on the strategy)
– solution	worked-out solution for the current exercise
– stepsremaining	number of remaining steps (based on the strategy)
– subtasks	returns a list of subtasks of the current task
meta-information	
– exerciselist	all supported exercise classes
– rulelist	all rules in an exercise class
– rulesinfo	detailed information about rules in an exercise class
– strategyinfo	information about the strategy of an exercise class

Table 2: Feedback services

in selecting or generating a task for a student. A domain reasoner is mainly used to support the inner loop of a tutoring system, in which it either *analyzes* the work of a student, or provides information about how to proceed, or what a solution to a task looks like. In the latter case, a domain reasoner *computes* a step for a task, or a solution for a task. Finally, a domain reasoner provides *meta-information* about itself: what kind of exercises are supported, what kind of steps can a student take, what kind of errors are recognized, etc. This section introduces feedback services offered by domain reasoners to tutoring systems. We introduce a number of basic services for the different components of a tutoring system, for the outer loop of the tutoring system, for the feedback and feedforward functionality of the inner loop, and for providing meta-information about a domain reasoner. Table 2 provides an overview of the feedback services that are introduced.

3.1. Services for the outer loop

The outer loop of a tutoring system selects tasks for a student. It may use information from the student model for this purpose, for example to select an exercise of a particular difficulty, but it may also ask for a random exercise in the domain. A domain reasoner may offer a *generate* service, which generates an exercise of a certain difficulty for which the domain reasoner provides support. This may be the starting term of an exercise from a (usually small) set of predefined examples in the domain reasoner, but it may also be a random exercise of the correct form and of the desired difficulty generated by a random generator. A domain reasoner may also offer an *examples* service for exporting the set of predefined examples.

3.2. Services for the inner loop

For the inner loop, a domain reasoner either analyzes a step, a final answer, or a worked-out solution for a task that was submitted by a student, or it computes a next step, a solution, or a worked-out solution.

The central service for analyzing a step is the *diagnose* service. For example, if a student rewrites the quadratic equation $x^2 - 4x + 3 = 0$ to $(x-3)(x-1) = 0$, the *diagnose* service from the domain reasoner responds with the diagnosis 'correct, but not finished'. It will also return `nice-factors (-3,-1)`, an abstract representation of the rewrite rule applied by the student. If the student submits $(x-4)(x-1) = 0$, the domain reasoner responds with the diagnosis 'incorrect'. The *diagnose* service also recognizes and reports applications of buggy rules. For example, if a student rewrites $(x+2)^2$ by $x^2 + 2^2$, the *diagnose* service returns the buggy rule diagnosis with the (buggy) rewrite rule `distr-square`. The *finished* service checks whether or not the exercise is solved.

The principal service for computing a next step towards a solution is the service *allfirsts*, which returns all possible next steps a student can take at the current stage of the exercise. For example, some of the possible next steps when solving the quadratic equation $x^2 - 4x + 3 = 0$ are finding the factors with rule `nice-factors (-3,-1)` and applying the quadratic formula with rule `abc (1,-4,3)`. Of course, a client learning environment interprets these rules, and presents them in a friendly way to users. The first rule can for example be presented as 'determine the factors of the quadratic expression', or, 'rewrite the current expression to $(x-3)(x-1) = 0$', depending on how a tutoring system wants to show the next step determined by a domain reasoner.

From the *allfirsts* service we obtain several derived services. The service *onefirst* returns only one next step in an exercise by selecting one of the alternatives. The step is determined from the steps returned by *allfirsts* by imposing an order on the steps. For example, the next step when solving the quadratic equation $x^2 - 4x + 3 = 0$ returned by *onefirst* would be `nice-factors (-3,-1)` because this rule is preferred over using the quadratic formula. The list of steps of a worked-out solution is provided by a *solution* feedback service. For example, a complete solution to the same exercise consists of a list of steps:

$$\begin{array}{ll} & x^2 - 4x + 3 = 0 \\ = & \quad \{\texttt{nice-factors (-3,-1)}\} \\ & (x-3)(x-1) = 0 \\ = & \quad \{\texttt{product-zero}\} \\ & x = 3 \vee x = 1 \end{array}$$

The *solution* service returns a stepwise solution for a particular exercise. If a client learning environment wants to show one or more worked-out solutions, it can use the *examples* service to obtain one or more exercises, and the *solution* service to turn these into worked-out solutions. Based on the solution, we can also calculate the number of remaining steps with the *stepsremaining* service, for example to show a progress indicator.

Some learning environments, such as MathPert [3], let a student select a rule to be applied to an expression, instead of rewriting an expression directly. To support performing such an interaction, we use a service *apply*, which applies a rewrite rule to a selected term.

When solving a quadratic equation such as $4 - 2x^2 = 4x + 6$, a student first has to move all terms to the left, then divide by -2, factorize the resulting expression, and solve the resulting linear equations. The *subtasks* service splits a task into such subtasks, and a learning environment can use the *subtasks* service to offer smaller tasks if a student cannot yet solve a complete task.

3.3. Meta-information about a domain reasoner

Domain reasoners have to publish meta-information about the type of exercises they support, the strategy that is used for solving a task, and the rewrite rules. Learning environments use this information to discover which functionality is supported by the domain reasoner. For instance, they can check whether all rewrite rules have a user-friendly translation that is presented to the students.

The *exerciselist* service returns all exercise classes that are supported by the domain reasoner. An explanation of the strategy of a domain reasoner is provided by the service *strategyinfo*. The service *rulelist* specifies which rules are allowed to be used when solving an exercise in the domain, and which buggy rules are recognized by the domain reasoner. The service *rulesinfo* provides more information about these rules, such as a textual description of a rule, and possibly a formal mathematical property (FMP, which is part of the OpenMath standard [11]) that is derived from the rule.

3.4. Conclusion

The complete list of services that can be used by learning environments is given in Table 2. Appendix A presents a full example of a sequence of interactions between a learning environment and a domain reasoner. The set of feedback services is not complete. Services such as supporting fill-in-the-blank exercises or assessing stepwise solutions are easily built on top of the existing services.

4. Feedback types in the literature

In this section we relate the services introduced in the previous section to the various kinds of feedback types discussed in the literature. Narciss [41] gives an overview of the literature on feedback types. We will have a look at the feedback types related to solving tasks. On a global level, the literature distinguishes the following feedback types.

- *Knowledge of performance* (KP) provides learners with summative feedback after they have responded to a set of tasks. This feedback contains information on the achieved performance level for this set of tasks (e.g., the percentage of correctly solved tasks).

- *Knowledge of result/response* (KR) provides learners with information about the correctness of their actual response (e.g., correct/incorrect).
- *Knowledge of the correct response* (KCR) provides the correct answer to the given task.
- *Answer-until-correct* (AUC) feedback provides KR and offers the opportunity of further attempts with the same task until the task is answered correctly.
- *Multiple-try feedback* (MTF) provides KR and offers the opportunity of a limited number of further attempts with the same task.
- *Elaborated feedback* (EF) provides additional information besides KR or KCR.

Some of these feedback types are related to the outer loop of a tutoring system (KP, AUC, MTF), and some feedback types can be produced by a domain reasoner (KR, KCR, some components of EF). Note that KP fundamentally depends on KR.

Knowledge of response feedback checks whether or not a step of a student is correct or incorrect, and whether or not an exercise is solved correctly. The domain reasoner offers the *diagnose* service for checking correctness of a step, and the *finished* service for determining whether or not an exercise is correctly solved. Knowledge of the correct response involves giving the complete solution to an exercise, offered by the *solution* service.

The answer until correct and multiple try feedback types are to a large extent dealt with in the outer loop of a tutoring system. The only feedback service required from the domain reasoner for these feedback types is the *finished* service, which checks whether or not a response can be accepted as an answer.

4.1. Categories of elaborated feedback

Narciss distinguishes five categories of elaborated feedback: *Knowledge about task constraints* (KTC), *Knowledge about concepts* (KC), *Knowledge about mistakes* (KM), *Knowledge about how to proceed* (KH), and *Knowledge about metacognition* (KMC).

Knowledge about task constraints involves hints or explanations on the type of task, on task-processing rules, on subtasks, and on task requirements. The type of task is usually specified in the task description, and is used to decide which domain reasoner to use for the task. For example, if the task is to solve a quadratic equation, the tutoring system will call the domain reasoner for quadratic equations to check correctness of a student step, show worked-out solutions, etc. The task processing rules are the rules that can be used to solve the task. This depends on the domain reasoner, and the feedback service *rulelist* specifies which rules are allowed to be used when solving an exercise in the domain. Since a subtask is by definition part of a complete task, a domain reasoner knows which subtasks need to be performed to complete a task. For example, the subtasks of solving a quadratic equation are: turn an equation into the form $ax^2 + bx + c = 0$, factorize the left-hand side of the equation, or apply the quadratic formula, and solve the linear equations thus obtained. A list of subtasks of a task is obtained via a service *subtasks*.

Knowledge about concepts involves hints or explanations of technical terms, examples illustrating a concept, etc. These components are usually references to material in the expert knowledge module.

Knowledge about mistakes involves reporting the number of mistakes, the location of a mistake, hints or explanations of the type of errors and the sources of errors. To provide this kind of information, the *diagnose* feedback service should report more than just correct/incorrect, but also location information, information about whether an error is a parsing error ($x^2-4x+=0$), a 'domain' error, such as rewriting $x^2-4x+3=0$ to $(x-4)(x-1)=0$, or possibly another kind of error. For example, a common kind of error when solving equations is forgetting a solution.

Knowledge about how to proceed involves bug-related hints for error correction, hints or explanations of task-specific strategies, hints or explanations of task-processing steps, guiding questions, or worked-out solutions. The *diagnose* service also recognizes and reports applications of buggy rules. An explanation of the strategy for solving a particular task is provided by the feedback service *strategyinfo*. As described earlier, the feedback service *rulelist* specifies all rules allowed to be used in a solution. The feedback services *onefirst* and *allfirsts* return a hint about the next step in an exercise, and a hint about all steps that are applicable according to the strategy at the current stage of the exercise, respectively. A guiding question can be derived from the hint returned by *onefirst*. For example, if *onefirst* returns `nice-factors (-3,-1)`, the guiding question translation might be 'try to determine the factors of the quadratic polynomial', a textual description might be 'use the factors -3 and -1 to rewrite the expression', and the bottom-out hint might be 'rewrite the expression to $(x-3)(x-1)=0$'. So the rule returned by the domain reasoner might be translated in various ways by the tutoring system to provide the kinds of feedback in the KH category. Finally, worked-out solutions are obtained by using the service *examples* to obtain example exercises, and *solution* to obtain their worked-out solutions.

Knowledge about metacognition involves hints or explanations of metacognitive strategies and metacognitive guiding questions. This kind of feedback is not provided by domain reasoners.

5. Internal components of an exercise class

What is a mathematical exercise? What do we need to know about an exercise to analyze student interactions, and to give feedback? How can we provide the feedback services for a class of exercises?

The exercises we consider consist of an expression and a goal (e.g., factorize x^2-3x+2). The expression that a student has to rewrite can be any kind of

component	*description*
strategy	rewrite strategy that specifies how to solve an exercise
rules	possible rewrite steps (including buggy rules)
equivalence	tests whether two terms are semantically equivalent
similarity	tests whether two terms are (nearly) the same
suitable	identifies which terms can be solved by the strategy
finished	checks whether a term is in a solved form
exercise id	identifier that uniquely determines the exercise class
status	stability of the exercise class
parser	parser for terms
pretty-printer	pretty-printer for terms (inverse of parsing)
navigation	supports traversals over terms
rule ordering	tiebreaker when more than one rule can be used
examples	list of examples, each with an assigned difficulty
random generator	generates random terms of a certain difficulty
test generator	generates random test cases (including corner cases)

Table 3: Exercise components providing domain-specific knowledge

expression, such as a polynomial, an equation or inequation (solve $x^2 + 2 = 3x$, with $x = 1$ or $x = 2$ as the solution), a system of linear equations, a matrix (for performing Gaussian elimination), a logical proposition (rewrite $\neg(p \wedge q) \vee r$ to disjunctive normal form), and many more. We abstract over the type of the expressions that appear in an exercise. Besides the type of the expression, more variation can be found between exercises. In this section we discuss how to encapsulate this variation. The variation is captured by making the components that together define an *exercise class* explicit. An exercise class contains all the domain-specific information for one particular type of exercise.

The design of our domain reasoners providing feedback services is based on two assumptions: all feedback is calculated automatically from a high-level description of an exercise class, and this calculation is generic (domain independent). With domain independence we mean that the design works for exercises in mathematics, logic, linear algebra, etc. The high-level description specifies how to solve a class of exercises, for example, the class of quadratic equations, and not just one specific equation. Note that for most classes of exercises, there are many possible solutions for one particular exercise, and each of these solutions must be taken into account when calculating feedback. For instance, consider the various ways in which the equation $x^2 + 4(x - 3) = 0$ can be solved. Even with extensive support for multiple solution paths for an exercise, we cannot anticipate all (correct) student intermediate answers. It is therefore important that feedback is still available after an unexpected step by a student.

In the remainder of this section we discuss the components that are needed for an exercise class. Table 3 presents an overview of these components. We start with explaining rewriting strategies, which is the central component of an exercise class. We conclude the section by showing that the components encapsulate the variation in exercise classes and that the feedback services can be implemented with these components.

5.1. Rewrite strategies for exercises

A rewrite strategy specifies sequences of rewrite steps (rules) for solving a class of exercises [32, 31], and models how a domain expert (such as a teacher) would solve an exercise. Thus it is foremost the rewrite strategy that contains the intelligence that is needed for solving exercises and for generating helpful feedback. The sequences of a rewrite strategy can be expressed as the sentences of a context-free grammar, in which the rewrite steps are used as the terminal symbols of the grammar. Viewing a rewrite strategy as a context-free grammar is useful because it allows us to take advantage of parsing techniques for the calculation of feedback. For example, tracking intermediate steps made by a student can now be formulated as a parsing problem.

We follow the combinator approach for specifying a rewrite strategy, which means that simple strategies can be composed into larger strategies for more complex exercises by using combinators. At the basic level, a rewrite strategy consists of a single rewrite step. Combinators for composing strategies include combinators for sequence (‘do this before that’), choice (‘either do this or that’), biased choice (‘try to do this, or else do that’), repetition (‘repeat this’), etc. A precise formalization of the strategy combinators, including the interleaving of strategies, is presented by Heeren et al. [31, 30]. The strategy combinators have been implemented as an embedded domain-specific language [33].

Inspired by the corresponding concepts from parser technology for context-free grammars, the functions *empty* and *firsts* provide the central functionality on rewrite strategies. The former tests whether or not the strategy is finished (i.e., is the empty sentence accepted), the latter returns the set of rewrite steps with which a sequence can start, together with the remainder of the strategy. These functions on strategies directly correspond with the *finished* and *allfirsts* feedback services that have been introduced in Section 3.

The compositional specification of strategies naturally supports the hierarchical structure of tasks and sub-tasks that can be found in many exercise domains. Labels can be attached to any sub-strategy (sub-task) for specializing the feedback generated by that part of the strategy.

5.2. Exercise classes

An exercise class contains all the components that are needed for calculating feedback. We will illustrate our design by presenting an exercise class for solving quadratic equations such as $x^2 - 4x + 3 = 0$. An exercise class has the following components:

- A *rewrite strategy* that specifies how an exercise can be solved. Strategies are constructed with the strategy combinators. The strategy for quadratic equations contains a sub-strategy that searches for factors, and knows how and when to apply the quadratic formula.

- *Rules* and common misconceptions described as *buggy rules* specify possible rewrite steps. Both types of rules, sound and buggy, can be applied to a term, and are used for recognizing a step made by the student. The rule set contains, by definition, all the rules that are used in the strategy. The rule set can also have rules that do not appear in the strategy, for instance because they are not supposed to be used. A buggy rule for polynomials is the incorrect distribution of an exponent over addition, $(x+y)^2 \not\Rightarrow x^2+y^2$.

- An *equivalence relation* for terms. With this relation we can test whether or not two terms are semantically equivalent. For example, $x^2-4x+3=0$, $(x-3)(x-1)=0$, and solution $x=3$ or $x=1$ are all equivalent. Equivalence relations are reflexive, symmetric, and transitive.

- Predicates *suitable* and *finished* on terms. The predicate suitable identifies which terms can be solved by the strategy of the exercise class. Suitable terms can be used as exercises for students. For example, the exercise class for solving quadratic equations can only handle equations of quadratic polynomials, and does not accept polynomials of a higher degree. The predicate finished, on the other hand, checks if a term is in a solved form (accepted as a final solution). For quadratic equations we check that all solutions have variable x on the left-hand side and a constant value on the right-hand side. What is considered a solved form may vary across exercise classes.

These components are the core of an exercise class: they have to be defined for each exercise class. Each exercise class must satisfy two properties that test the internal consistency of the components. Firstly, all non-buggy rules must be sound with respect to the equivalence relation. In other words, the result of applying a rule to a term should always be equivalent to the term itself. Because the rewrite strategy is composed from rules and applies the rules in a certain order, the strategy is also sound with respect to the equivalence relation. Secondly, the rewrite strategy of an exercise class should solve the exercise for all terms that satisfy the suitable predicate. In fact, the suitable and finished predicates act as the pre- and post-condition for the strategy. The predicates are a contract for the strategy [39], simplifying the correctness testing of an exercise class.

Defining an equivalence relation may introduce a technical challenge for more complex exercise classes, since equivalence of terms is undecidable for certain domains. Examples of such domains are exercise classes that involve higher-degree polynomials: in an exercise about computing derivatives, we may need equivalence of polynomials. Other examples are exercises about rewriting context-free grammars and equivalence in programming tutors. This problem can be circumvented in most cases: sometimes, the solution of an exercise is known a priori, which makes it easier to test for equivalence, in other cases the student is not expected (or allowed) to make rewrites that are not anticipated, resulting in a more restricted interpretation of equivalence.

5.3. Granularity in exercises

In mathematics, there are many subtly different ways in which an expression can be represented, e.g., $x^2 + 4x$ and $4x + x^2$ are easily seen to be semantically the same expression. Even though such expressions are not the same, from the perspective of a student it is reasonable to expect that these expressions can be interchanged. The domain of polynomials contains many such examples. In most domain reasoners, the order of terms and the placing of parentheses in a summation or multiplication is irrelevant (associativity and commutativity), and idiosyncratic expressions such as $x + (-5)$ are rewritten implicitly. These variations should not have an effect on the reported feedback. Depending on the level of the exercise and its intended audience, it might well be reasonable to automatically perform simple calculations with constants, to simplify square roots (e.g., replace $\sqrt{8}$ by $2\sqrt{2}$), or to introduce exponents ($x \cdot x$ versus x^2). In a domain reasoner for adding fractions, however, it would be strange to perform the addition automatically.

Which terms are considered the same, in the sense that they are interchangeable, depends on the exercise class. We therefore introduce a second equivalence relation as a component of an exercise class. Besides the *equivalence* relation for semantic equivalence, we introduce the *similarity* relation for denoting syntactic similarity of terms. By carefully considering which terms are similar to each other and which are not, we are in fact selecting the granularity of the steps in an exercise class.

The similar relation should interact with the other components of an exercise class in a prescribed way: three more properties can be checked for each instance of an exercise class. Firstly, the similarity relation is a subset of the equivalence relation. In other words, terms that are similar must also be equivalent. Secondly, the result of applying a rule to a term must not be similar to the term itself. In this way, we can recognize the application of such a rule. Violating this property means that the rule does not respect the granularity of the exercise class. Thirdly, the predicates suitable and finished should give the same result for similar terms.

5.4. More components of exercise classes

In this subsection we present the remaining components of an exercise class: see Table 3 for an overview of all components. We need these components for the communication between the domain reasoner and the tutoring system, and for implementing the feedback services.

- Each exercise class has an *identifier* and a *stability*. A tutoring system uses the identifier to select a certain exercise class in case a domain reasoner supports more than a single exercise class.

- Terms can be *serialized* so that a domain reasoner can send terms to and receive terms from a tutoring system. The tutoring system must understand the format in which the terms are communicated since it typically has to present the terms to the student (in a nice way), and because it usually offers an editor for rewriting the term. Terms are serialized to strings: each exercise class has a *parser* and a *pretty-printer*. To make interoperating with a tutoring system as simple as possible, we also support the OpenMath standard [12, 11] for representing mathematical objects in XML.

- For stepwise solving an exercise, we often have to *navigate* through a term and select a sub-term that is rewritten. Navigation requires knowledge about the representation of a term. Because exercise classes abstract over the representation of terms, this information has to be supplied as a component of an exercise class. When we navigate over terms, we also get position information for rewrite steps ('which sub-term is changed?'), and we can use traversal strategies, such as bottom-up and top-down, in strategy specifications. Note that sub-terms can be of different types: for example, the term $x = 3 \vee x = 1$ contains a logical disjunction $(\ldots \vee \ldots)$, two equations ($x = 3$ and $x = 1$), and four expressions (x, 3, x, and 1). In a strongly-typed language for representing exercise classes, this implies that we also get strategies and rules within an exercise class that operate on terms of different types. We need some mechanism to lift strategies and rules to other types.

- A *rule ordering* orders the rules of an exercise class. The ordering is used as a tiebreaker when multiple next steps are computed by the *allfirsts* service, but only one step is reported or used, for instance, for implementing the *onefirst* feedback service. Worked-out solutions, generated by the *solution* feedback service, only represent one possible solution path, and this path is also selected by the rule ordering.

We consider generation of exercises, i.e., generating an initial term that a student has to rewrite, not a primary task of a domain reasoner. Nevertheless, it is valuable to be able to obtain exercises (initial terms) that belong to an exercise class, for several reasons. We describe the components for producing exercises and their motivation.

- Exercises are used as *examples* that demonstrate how the rewrite strategy of an exercise class works. An exercise class has a fixed set of examples that is used for generating documentation about the exercise class. These documentation pages have proven to be an effective way for communicating rewriting strategies to domain experts such as teachers.

- Even though tutoring systems can use exercises from their own expert knowledge module and submit these to a domain reasoner, many still use the *generate* feedback service to request an exercise of a certain level. We can specify a *randomized exercise generator* for an exercise class that is used to implement the *generate* feedback service. If such a generator is not present for an exercise class, then the feedback service returns one of the examples instead.

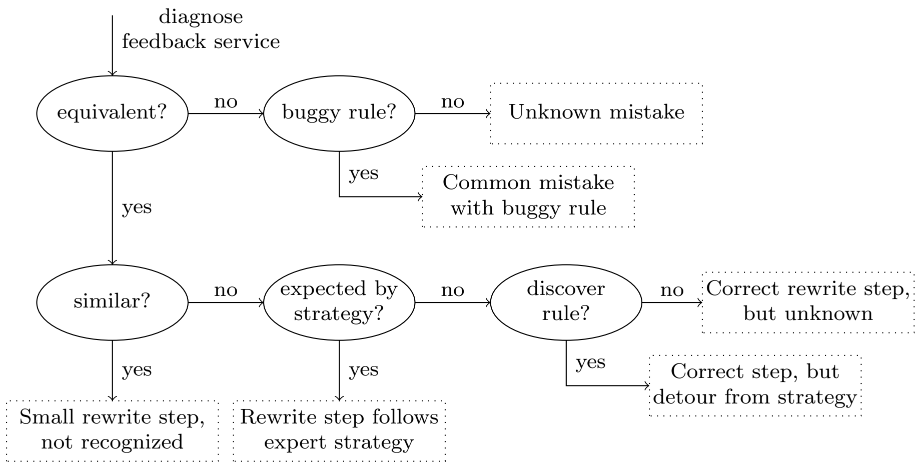


Figure 4: Structure of the *diagnose* feedback service

- We use a *test generator* for testing properties of exercise classes. When testing an exercise class we also want to check corner cases, so the terms generated for testing are not necessarily similar to the terms that are used as student exercises. Properties are tested with examples and random exercises if no test generator is defined for an exercise class.

Are there more components an exercise class should have? The components described in this section are sufficient for the domain reasoners and feedback services we have developed so far. In certain situations it is tempting to extend the exercise classes with more information that is often specific for a particular domain, risking a bloated interface for exercise classes. One such situation was for the Ask-Elle programming tutor [23], where we needed to specify additional properties for testing the correctness of a program. Such an extension is only meaningful for programming tutors, hence we decided against extending the interface. However, we do provide some backdoors in the actual implementation to deal with such extensions. The presented interface for exercise classes is coherent, it has non-trivial properties for testing consistency, and it is sufficient for implementing the feedback services.

5.5. Implementing the feedback services

The feedback services of Section 3 can be implemented with the components of an exercise class. The feedback services for computing a next step are directly implemented by the *empty* and *firsts* functions that are defined on rewrite strategies. Implementing the *diagnose* feedback service for analyzing a student step is more involved because several components of an exercise class are combined. The structure of the *diagnose* feedback service is presented in Figure 4.

We first check whether or not the submitted student term is equivalent to the term of the exercise. If not, the step made by the student is incorrect. The buggy rules are used to search for a common mistake. When a buggy is found, we can give a more detailed explanation why the step is incorrect. If the student term is equivalent we then check for similarity. When the terms are similar, the rewrite step is too small to be recognized. The student may have done some simplifications or some simple calculations that are too detailed for the exercise at hand. A learning environment can choose how to report this to the student. If the terms are equivalent but not similar, we use the rewrite strategy to compare the student step with the steps that are computed by strategy. If the term was not expected by the strategy, then the student deviated from the solution paths specified by the strategy. In this case, we can report that although the step is correct, a different step was suggested by the domain reasoner. The feedback message reported to the student for this diagnosis, however, should be phrased with care because the student step could be a shortcut (compared to the rewrite strategy) instead of a detour. For a correct step not expected by the strategy we try to discover which rule was used. Depending on the kind of exercise, a correct step with an unknown rule can be accepted or rejected: this can be decided by the learning environment that uses the feedback services. In a domain reasoner for practicing rewriting logical propositions, for example, it is reasonable that a student may only use a limited set of rewrite rules.

The diagram in Figure 4 does not contain the finished predicate for checking if the exercise is solved or not. All diagnoses for an equivalent step also contain the finished information for the student term.

6. Evaluation

In the previous sections we have described the design of a framework for developing domain reasoners that can be used by tutoring systems. So far, we have focused on dealing with variation in tutoring, feedback services, and exercise classes. The generic framework encourages the reuse of functionality that is shared by the domain reasoners and it constrains the design of a domain reasoner, making it easier to implement a reasoner for a new domain. In this section we evaluate the proposed design. First, some measured response times for feedback services are analyzed. Next, we discuss other non-functional quality attributes.

6.1. Response times for feedback services

Calculating feedback and hints must not delay the student that is trying to solve an exercise. The efficiency of the feedback services must therefore be within reasonable bounds. In his book on usability engineering, Jakob Nielsen [42] reports that 0.1 second is about the time limit for users to feel that the system is reacting instantaneously. For the implemented domain reasoners we have found that this limit imposes no constraints on the design. We have monitored

request	*number*
quadratic equations	57,750
linear equations	35,422
higher-degree equations	19,217
exerciselist (meta-information)	2,441
other	8,204
total	123,034

Table 4: Number of requests by DME (grouped by exercise)

request	*number*	*avg resp time (ms)*		
		linear	*quadratic*	*higher-degree*
diagnose	84,645	20.2	31.1	48.9
onefirst	14,297	34.1	30.2	27.6
solution	8,017	22.0	30.6	56.1
allfirsts	3,077	19.2	17.1	22.8
rulelist	1,768	21.5	17.1	15.9
findbuggyrules	417	20.0	–	–
rulesinfo	92	–	1291.6	–
examples	74	–	63.5	–
incorrect	2	–	–	–
total	112,389			

Table 5: Response times of services for polynomial equations in DME

response times of the domain reasoners to requests from the learning environments, and found that the feedback services are efficient enough for their intended use. The numbers support our own experience with the responsiveness of the feedback offered in the various learning environments.

We will have a closer look at the response times for two specific cases: a domain reasoner for polynomial equations used by the DME, and a domain reasoner for reaching disjunctive normal form in our own logic tool.

Polynomial equations used by DME. All requests from the DME are stored in a database. Most requests concern the three exercise classes for solving polynomial equations. Table 4 presents the number of requests between April 2009 and October 2013. It should be noted that these requests have been handled by different versions (upgrades) of the software, and that the requests come from students and teachers, but also from tool developers of DME.

Table 5 presents the average response times in milliseconds of the services for DME requests about polynomial equations. Most requests use the *diagnose* service, which is called at each step a user makes. Averages are only shown for more than 50 requests per exercise class. Response times were measured on the server-side, which means that the response times as perceived in DME were

request	*number*	*avg resp time (ms)*
diagnosetext	17,110	37.9
onefirsttext	8,287	37.9
generate	7,037	205.4
finished	1,718	28.4
solutiontext	1,081	73.1
solution	61	63.0
total	35,294	

Table 6: Response times of services for the logic tool

higher. The domain reasoners offering the feedback services were running on a virtual machine different from the server hosting DME: this setup is clearly not optimal (but nevertheless appears to be sufficient for its purpose).

The table shows average response times that are all below 100 milliseconds, except for the *rulesinfo* service. This service performs some additional computations to provide detailed meta-information about the rules. Meta-information about the supported exercise classes (service *exerciselist* with 2441 requests) takes 36.9ms on average. Observe that the services *diagnose* and *solution* take longer for more complex domains: the rewrite strategy becomes more complex for solving equations of a higher-degree, and more rewrite rules are involved (12 rules and 16 buggy rules for linear equations versus 32 rules and 31 buggy rules for higher-degree equations).

Appendix B presents additional information about the response times reported in Figure 5, including the distribution of the response times. A closer inspection of the response times for the *diagnose* service reveals that 1480 requests (1.75%) take longer than 100ms, and 76 requests (0.09%) take longer than 500ms. These delays are not caused by the calculation needed for dealing with the requests, but are most likely a result of the network connection.

Disjunctive normal form in the logic tool. Table 6 shows the response times for requests from our own logic tool. These services use a feedback script for reporting textual feedback (see Figure 3). The logic tool uses the *generate* service for generating random exercises for some level (easy, medium, or difficult). After generating a randomized proposition, the number of steps needed for solving is calculated using the rewrite strategy. If this number is not appropriate for the level, another proposition is tried. This approach explains the response time for the *generate* service that might be higher than expected.

Monitoring response times is a good guideline for the feasibility of new feedback services, as well as outlining the limitations of existing services. For example, we could add a search depth to the *diagnose* feedback service for classifying more intermediate answers by searching for combinations of rules. In practice, however, the search space quickly becomes too large to explore efficiently. This search is part of the plan recognition problem, which is known to be computationally very difficult [2].

6.2. Non-functional quality attributes

We now discuss other non-functional quality attributes of the design for providing feedback services: scalability, deployability, testability, traceability, and interoperability.

Scalability and deployability. Domain reasoners must be scalable and be able to handle many users simultaneously. Just the DME, one of the clients of our domain reasoners, has over 10,000 yearly users. Designs based on the stateless client-server style are known to scale well [21]. Another advantage of the client-server style is that learning environments can use local copies of the domain reasoners to remove the dependency on externally hosted services. DME and Math-Bridge successfully deployed our services on servers in the USA and Germany, respectively.

Since we are hosting the domain reasoners on a server, it is easy to deploy a new version of a domain reasoner beside the older version. This is useful when a domain reasoner is under development and a strategy has to be refined to incorporate suggestions by domain experts.

Testability. Writing high-quality software without defects is an important requirement in many situations. In educational software, where users are supposed to learn from using the software, it is especially important that the reported feedback is correct and helpful. There are many different approaches to develop software with high internal quality. We discuss the contributions of the framework to the internal quality of the domain reasoners.

To start with, the most error-prone parts of a domain reasoner are handled by the framework and are thus reused for all domain reasoners, such as the implementation of the feedback services and the low-level request-reply messages, in different formats, that are exchanged between the client and the server. The central component of an exercise class is the rewrite strategy, from which feedback is calculated automatically. The semantics of the strategy language has been formalized [31] and this formalization helps to get the calculations based on a strategy correct. In Section 5 we have formulated a number of properties for exercise classes. These properties help to test the consistency of the implemented components for an exercise class.

Traceability. Every request-reply pair that is processed by our domain reasoners is stored in a database. With this data we can analyze student behavior, for instance to determine what kind of feedback is reported, how effective the feedback is, or to search for common mistakes and to formulate buggy rules for these mistakes. For example, Lodder et al. [37] have reported on using the information in request-reply pairs for improving the logic tool. In our own tutors, we include an optional student-id in the request-reply pairs so that we can later single out sessions of individual students. The stored data is used for analyzing the frequency and volume of requests (from a specific client, or the overall volume), and for monitoring the response times of the feedback services. A stateless client-server style makes it easy to store all interactions for further analysis.

Interoperability. The feedback services are light-weight and easy to use. Requests and replies are based on popular standards, including OpenMath for representing mathematical objects. This makes it relatively easy for learning environments to start using our feedback services, and indeed, several external learning environments now have a connection with our domain reasoners.

7. Lessons learned

We have been working on developing domain reasoners for more than six years[1]. The domain reasoners and the design of the feedback services have evolved from a research prototype into the current state, in which the domain reasoners provide feedback to several learning environments for a variety of courses. In this section we document the lessons learned. We reflect on some of our design decisions, describe limitations, and explore alternatives.

7.1. Shared knowledge

Ideally, the learning environment (client) and domain reasoner (server) are loosely coupled and hide most of their details for the other. Learning environments should be largely unaffected by changes in the domain reasoner. In practice, however, both systems must share a considerable amount of knowledge about the details of feedback services, exercise classes, rewrite rules, etc. For example, the abstract representations of the rewrite rules are introduced by the domain reasoner and must be known by the learning environment for reporting feedback to students. Modifying the set of rewrite rules in a domain reasoner thus affects the quality of feedback in a learning environment. For domain reasoners that are actively used by (multiple) learning environments, backward compatibility of the provided feedback services and exercise classes becomes an important consideration.

A simple solution to improve the stability of feedback services and exercise classes is to allow different versions of a domain reasoner to be deployed for different learning environments. This provides more control when upgrading certain parts of a domain reasoner for one environment while leaving the domain reasoner untouched for another environment (e.g., because it is used in the class room). Alternatively, we can construct dedicated wrappers that translate requests using an outdated knowledge representation into the latest knowledge representation, and similarly for the responses. One lesson learned is that the feedback services should be designed with forward and backward compatibility in mind where possible.

Shared knowledge between learning environments and domain reasoners cannot be avoided for tasks that are, figuratively speaking, at the boundary of the learning environment and the domain reasoner component, or for tasks that can be done at either side. The exact formulation of textual feedback messages for

[1]See `http://ideas.cs.uu.nl/www/` for an overview of our work on domain reasoners, and `http://hackage.haskell.org/package/ideas` for the software package.

a certain diagnosis, for example, can be done by the learning environment or by the domain reasoner. Assigning the responsibility for feedback texts to the learning environment has the advantage that feedback messages can be better tailored for their intended use in tutoring. Some learning environments have extended their authoring environment to edit the feedback messages connected to rewrite rules. The authoring environment of the DME also lets teachers customize how much feedback is available for a particular exercise (see Chapter 5 in [5]). A disadvantage is that detailed information about the rewrite rules and strategies is required. It must be clear which rules and sub-strategies are used in an exercise class, which is exactly why we have defined feedback services that provide meta-information about an exercise class. The feedback scripts, shown in Figure 3, are an alternative solution with server-side feedback texts.

Some advanced learning environments maintain a student model to track the progress of a student and to adapt the system to the level of the student. A domain reasoner computes detailed information about correct and incorrect applications of rewrite rules and sub-tasks, and this information is highly relevant for updating the student model. Updating the student model with information from the domain reasoner requires a shared understanding of the exercise domain. Information in the student model could be used for exercise generation or selection. If this task is performed by the domain reasoner, then it must have access to the student model.

For all our exercise classes, rewrite strategies, and rules we automatically generate extensive documentation to make it easier for developers of learning environments and content authors to see which concepts are available. For mathematical domain reasoners we started using a standard taxonomy [45] to classify the exercise classes, rewrite strategies, and rules. Math-Bridge has used this taxonomy to extend the exercises offered by our domain reasoners with meta-information used by Math-Bridge's student model.

7.2. REST architectural style

The Representational State Transfer (REST) architectural style, first described by Fielding and Taylor [21], introduces a set of principles that guide the design of web services, or, in our case, the feedback services provided by domain reasoners. The REST architectural style is based on a stateless client-server architecture, in which client and server communicate by sending request and response messages. The key concept of REST is a resource, which can be any kind of potentially useful information. The resources that can be identified in our domain reasoners are the parts that model domain-specific knowledge, including exercise classes, rewrite strategies, and rewrite rules. We briefly discuss how the REST principles can be used to further improve the design of the feedback services.

A key principle of the REST style is to assign identifiers to resources that are relevant. The default way for accessing a resource is by way of Uniform Resource Identifiers (URI). In the current design this principle is followed only to a certain extent, and URIs are not used to their full potential. At the moment, clients are exposed to the underlying technology for handling requests, namely CGI binaries.

Another principle is that resources can have multiple representations, including representations that are targeted at humans (e.g., HTML) instead of being easy to process by a machine. This principle could potentially unify the feedback services and the documentation that is generated for the resources. A final REST principle suggests to link resources together so that it becomes easier to explore the available resources, and to use hypermedia as the engine of application state [20]. Addressing the REST principles in the design of our domain reasoners is future work.

7.3. Configuration and adaptability

A final consideration in the design of domain reasoners is how easy it is to adapt or configure a domain reasoner. From discussions with teacher, developers of learning environments, and students we have concluded that a 'one size fits all' approach does not work well for domain reasoners. These groups of users have different requirements about customizing exercises. A teacher, for instance, may want to influence which rewrite rules may be used (and which not), which approach should be taken for solving an exercise, how fractions are presented (e.g., $2\frac{1}{3}$ versus $\frac{7}{3}$), what the granularity of an exercise is, and which terms are in a solved form (e.g., is the improper fraction $\frac{7}{3}$ accepted as a solution or not). Hence, it should be possible to adapt domain reasoners to specific needs [29] without changing the software.

Mathematical learning environments usually offer topics incrementally, building upon prior knowledge. Following Beeson's principles [4] of *cognitive fidelity* (the software solves the problem as a student does) and *glassbox computation* (you can see how the software solves the problem), domain reasoners should be organized with the same incremental and layered organization. We are following such an organization for the concepts that model knowledge, such as rewrite strategies and rules. It should be possible to inspect the internal structure of these concepts, and their relation to other concepts.

At the moment, our domain reasoners have limited mechanisms for configuration and adaptation. For example, we have proposed a simple XML language for configuring rewrite strategies [20]. Nevertheless, it requires specialized knowledge to define a domain reasoner, or to modify an existing one. By nature, strategies are often complex artifacts.

8. Related work

Since developing a learning environment such as an intelligent tutoring system is a major undertaking, developing a software architecture for such a system is important. Nwana [44] gives a nice overview of the work in the 1980s on developing an architecture for intelligent tutoring systems. The resulting architecture, which divides an intelligent tutoring systems into four conceptual components, is still used today, see for example the recent Advances on Intelligent Tutoring Systems [43]. The architecture is agnostic of the approach an intelligent tutoring system takes to representing expert knowledge: both model-tracing tutors [2] and constraint-based tutors [40] are described using the same architecture. The architecture has been used to develop stand-alone intelligent tutoring systems as well as web-based solutions [1]. This paper deals with an architecture for one of the components of intelligent tutoring systems, namely the expert knowledge module. Our current implementation of this architecture uses a model-tracing approach to calculate feedback, but using a constraint-based approach to calculate feedback is also possible. We expect that observations from previous comparisons [34], such as that it is often easier to develop constraint-based tutors, and that model-tracing makes it easier to guide a student towards a goal, apply to the development of feedback webservices too.

Brusilovsky [8] and Ritter, Koedinger, and colleagues [48, 47] introduce an integration-oriented or component-based architecture for intelligent tutoring systems. The main goal of these architectures is to use tutoring components in other software, or to compose intelligent tutoring systems out of components. The tutor agent described by Ritter and Koedinger performs the services our webservices offer. Our approach can be viewed as offering the various actions a tutor agent performs as webservices. For the same reason why it is useful to introduce a component-based architecture for an intelligent tutoring system, it is useful to describe the various actions of a tutor agent as separate webservices.

Several authors have discussed the advantages of using web-based systems for intelligent tutoring [10, 46]. Patvarczki et al. [46] discuss the performance of web-based intelligent tutoring systems, in particular the time it takes to create a page for a student. Our work provides performance information about components of these pages.

Goguadze designs a query language and a set of domain reasoning queries to be used for diagnosing interactions of students working on interactive exercises in a learning environment [26, 25, 24]. The queries are easily translated in our webservices, and our webservices can be viewed as a webservice based implementation of the query language.

Zinn [56, 57] also develops domain reasoners. He doesn't discuss the architecture of domain reasoners, or separate the various kinds of feedback into separate services.

9. Conclusions

We have shown how domain reasoners can be developed that offer feedback services for stepwise exercise. By using these services, learning environments can better support students solving stepwise exercises. We have related the feedback services to the types of feedback discussed in the literature. A domain reasoner offers functionality to obtain feedback about a particular class of exercises. For developing a domain reasoner, we introduced the concept of an exercise class, which is a record with several components that are necessary for calculating feedback, such as equivalence of two terms, and suitability of an exercise. The most important among these components is a rewrite strategy that describes how an exercise in the class of exercises is solved. The strategy is used to diagnose a student step, and to calculate a next step or a complete solution for an exercise. We have discussed the non-functional quality attributes of domain reasoners, and shown how they are implemented such that they are efficient, scalable, testable, etc.

In the future we want to experiment with domain reasoners for other domains. We are working on a domain reasoner for imperative programming, similar to our domain reasoner for functional programming. An interesting problem in programming tutors is 'blaming': which part of the program is responsible for an error. We want to use property checking and contract checking to improve the feedback in programming tutors. We are also developing a domain reasoner for a serious game for practicing communication skills for pharmacists, doctors, veterinarians, and psychologists. Furthermore, we are investigating the use of domain reasoners for implementing the AI for real-time video games [28]. On a more conceptual level, we are looking at how we can offer a highly expressive language for formulating rewrite strategies, yet keep efficient execution of our services.

Acknowledgements. Alex Gerdes has helped in developing the services and software discussed in this paper. Many students contributed to parts of our domain reasoners. The developers of the learning environments that use our feedback services have provided valuable feedback about the form and content of our services. The anonymous reviewers helped improving the content and presentation of the paper.

References

[1] Sherman R. Alpert, Mark K. Singley, and Peter G. Fairweather. Deploying intelligent tutors on the web: An architecture and an example. *International Journal of Artificial Intelligence in Education*, 10:183–197, 1999.

[2] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. Cognitive tutors: lessons learned. *The Journal of the Learning Sciences*, 4(2):167–207, 1995.

[3] Michael J. Beeson. A computerized environment for learning algebra, trigonometry, and calculus. *International Journal of Artificial Intelligence in Education*, 1:65–76, 1990.

[4] Michael J. 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.

[5] Christian Bokhove. *Use of ICT for acquiring, practicing and assessing algebraic expertise.* PhD thesis, Utrecht University, 2011.

[6] Russell Bradford, James H. Davenport, and Christopher J. Sangwin. A comparison of equality in computer algebra and correctness in mathematical pedagogy. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen,

and Stephen M. Watt, editors, *Intelligent Computer Mathematics*, volume 5625 of *Lecture Notes in Computer Science*, pages 75–89. Springer-Verlag, 2009.

[7] Peter Brusilovsky. The construction and application of student models in intelligent tutoring systems. *Journal of Computer and Systems Sciences International*, 32(1), 1994.

[8] Peter Brusilovsky. Intelligent learning environments for programming: The case for integration and adaption. In J. Greer, editor, *Proceedings of AI-ED '95: the World Conference on Artificial Intelligence in Education*, pages 1–7, 1995.

[9] Peter Brusilovsky. Adaptive hypermedia: From intelligent tutoring systems to web-based education. In *Proceedings of ITS '00: the 5th International Conference on Intelligent Tutoring Systems*, pages 1–7. Springer-Verlag, 2000.

[10] Peter Brusilovsky and Christoph Peylo. Adaptive and intelligent web-based educational systems. *International Journal of Artificial Intelligence in Education*, 13(2-4):159–172, 2003.

[11] Olga Caprotti, D. P. Carlisle, and Arjeh Cohen (Eds). *The OpenMath Standard*. OpenMath Consortium, August 1999.

[12] Olga Caprotti, Arjeh Cohen, Hans Cuypers, and Hans Sterk. OpenMath technology for interactive mathematical documents. In Jonathan Borwein, Maria H. Morales, Konrad Polthier, and José F. Rodrigues, editors, *Multimedia Tools for Communicating Mathematics*, pages 51–66. Springer-Verlag, 2002.

[13] Hamid Chaachoua, Jean-François Nicaud, Alain Bronner, and Denis Bouhineau. APLUSIX, a learning environment for algebra, actual use and benefits. In Mogens Niss, editor, *ICME 10: 10th International Congress on Mathematical Education*, 2004. Retrieved from `http://hal.archives-ouvertes.fr/docs/00/19/03/93/PDF/Chaachoua-h-2004.pdf`, April 2013.

[14] Arjeh Cohen, Hans Cuypers, Dorina Jibetean, and Mark Spanbroek. Interactive learning and mathematical calculus. In Michael Kohlhase, editor, *Mathematical Knowledge Management, 4th International Conference, MKM 2005, Bremen, Germany, July 15-17, 2005, Revised Selected Papers*, volume 3863 of *Lecture Notes in Computer Science*, pages 330–345. Springer-Verlag, 2005.

[15] Arjeh Cohen, Hans Cuypers, Ernesto Reinaldo Barreiro, and Hans Sterk. Interactive mathematical documents on the web. In Michael Joswig and Nobuki Takayama, editors, *Algebra, Geometry and Software Systems*, pages 289–306. Springer-Verlag, 2003.

[16] European Commission. Opening up education: Innovative teaching and learning for all through new technologies and open educational resources. Communication from the commission to the European parliament, the council, the European economic and social committee and the committee of the regions, September 2013.

[17] Albert T. Corbett, Kenneth R. Koedinger, and John R. Anderson. Intelligent tutoring systems. In M. Helander, T. K. Landauer, and P. Prahu, editors, *Handbook of Human-Computer Interaction, Second Edition*, pages 849–874. Elsevier Science, 1997.

[18] Hans Cuypers, Jan-Willem Knopper, and Hans J.M. Sterk. MESS: the MathDox Exercise System. In *e-Proceedings of the 6th JEM Workshop*, pages 1–12, 2009.

[19] Michel Doorman, Paul Drijvers, Peter Boon, Sjef van Gisbergen, and Koeno Gravemeijer. Design and implementation of a computer supported learning environment for mathematics. In *Earli 2009 SIG20 invited Symposium Issues in designing and implementing computer supported inquiry learning environments*, 2009.

[20] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[21] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[22] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Sylvia Stuurman. Feedback services for exercise assistants. In D. Remenyi, editor, *ECEL*, pages 402–410. Acad. Publ. Ltd., 2008.

[23] Alex Gerdes, Johan Jeuring, and Bastiaan Heeren. An interactive functional programming tutor. In T. Lapidot, J. Gal-Ezer, M.E. Caspersen, and O. Hazzan, editors, *Proceedings of ITICSE 2012: the 17th Annual Conference on Innovation and Technology in Computer Science Education*, pages 250–255. ACM Press, 2012.

[24] George Goguadze. Representation for interactive exercises. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 of *Lecture Notes in Computer Science*, pages 294–309. Springer-Verlag, 2009.

[25] George Goguadze. *ActiveMath - Generation and Reuse of Interactive Exercises using Domain Reasoners and Automated Tutorial Strategies*. PhD thesis, Universität des Saarlandes, Germany, May 2011.

[26] George Goguadze and Erica Melis. Combining evaluative and generative diagnosis in ActiveMath. In *Proceedings of the 2009 conference on Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modelling*, pages 668–670. IOS Press, 2009.

[27] Barbara Grabowski, Susanne Gäng, Jörg" Herter, , and Thomas Köppen. Mathcoach and Laplacescript: Advanced exercise programming for mathematics with dynamic help generation. In *Proceedings of the ICL2005 Workshop, at International Conference on Interactive Computer Aided Learning ICL, Villach, Austria*, 2005.

[28] Tom Hastjarjanto, Johan Jeuring, and Sean Leather. A DSL for describing the artificial intelligence in real-time video games. In *Proceedings GAS 2013: the 3rd International Workshop on Games and Software Engineering*, pages 8–14. IEEE, 2013.

[29] Bastiaan Heeren and Johan Jeuring. Adapting mathematical domain reasoners. In *Proceedings of MKM 2010: the 9th International Conference on Mathematical Knowledge Management*, volume 6167 of *LNAI*, pages 315–330. Springer-Verlag, 2010.

[30] Bastiaan Heeren and Johan Jeuring. Interleaving strategies. In *Proceedings of MKM 2011: the 10th International Conference on Mathematical Knowledge Management*, volume 6824 of *LNAI*, pages 196–211. Springer-Verlag, 2011.

[31] Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.

[32] Bastiaan Heeren, Johan Jeuring, Arthur van Leeuwen, and Alex Gerdes. Specifying strategies for exercises. In *Proceedings of MKM 2008: the 7th International Conference on Mathematical Knowledge Management*, volume 5144 of *LNAI*, pages 430–445. Springer-Verlag, 2008.

[33] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), 1996.

[34] Viswanathan Kodaganallur, Rob R. Weitz, and David Rosenthal. A comparison of model-tracing and constraint-based intelligent tutoring paradigms. *International Journal of Artificial Intelligence in Education*, 15(2):117–144, 2005.

[35] Josje Lodder and Bastiaan Heeren. A teaching tool for proving equivalences between logical formulae. In Patrick Blackburn, Hans Ditmarsch, Mara Manzano, and Fernando Soler-Toscano, editors, *Tools for Teaching Logic*, volume 6680 of *Lecture Notes in Computer Science*, pages 154–161. Springer-Verlag, 2011.

[36] Josje Lodder, Johan Jeuring, and Harrie Passier. An interactive tool for manipulating logical formulae. In M. Manzano, B. Pérez Lancho, and A. Gil, editors, *Proceedings of the Second International Congress on Tools for Teaching Logic*, 2006.

[37] Josje Lodder, Harrie Passier, and Sylvia Stuurman. Using IDEAS in teaching logic, lessons learned. In *International Conference on Computer Science and Software Engineering*, volume 5, pages 553–556, 2008.

[38] Erica Melis and Jörg Siekmann. ActiveMath: An intelligent tutoring system for mathematics. In *ICAISC*, volume 3070 of *Lecture Notes in Computer Science*, pages 91–101. Springer-Verlag, 2004.

[39] Bertrand Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, Englewood Cliffs, N.J., 1991.

[40] Antonija Mitrovic, Brent Martin, and Pramuditha Suraweera. Intelligent tutors for all: The constraint-based approach. *IEEE Intelligent Systems*, 22(4):38–45, 2007.

[41] Susanne Narciss. Feedback strategies for interactive learning tasks. In J.M. Spector, M.D. Merrill, J.J.G. van Merriënboer, and M.P. Driscoll, editors, *Handbook of Research on Educational Communications and Technology*. Mahaw, NJ: Lawrence Erlbaum Associates, 2008.

[42] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1993.

[43] Roger Nkambou, Jacqueline Bourdeau, and Riichiro Mizoguchi, editors. *Advances in Intelligent Tutoring Systems*, volume 308 of *Studies in Computational Intelligence*. Springer-Verlag, 2010.

[44] Hyacinth S. Nwana. Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 4(4):251–277, 1990.

[45] Mathematical Sciences Conference Group on Digital Educational Resources. Core subject taxonomy for mathematical sciences education, april 29, 2005. Retrieved from `http://people.uncw.edu/hermanr/MathTax/`.

[46] Jozsef Patvarczki, Shane F. Almeida, Joseph E. Beck, and Neil T. Heffernan. Lessons learned from scaling up a web-based intelligent tutoring system. In *Proceedings of ITS '08: the 9th international conference on Intelligent Tutoring Systems*, pages 766–770. Springer-Verlag, 2008.

[47] Steven Ritter, Peter Brusilovsky, and Olga Medvedeva. Creating more versatile intelligent learning environments with a component-based architecture. In *Proceedings of ITS '98: the 4th International Conference on Intelligent Tutoring Systems*, pages 554–563. Springer-Verlag, 1998.

[48] Steven Ritter and Kennth R. Koedinger. An architecture for plug-in tutor agents. *International Journal of Artificial Intelligence in Education*, 7(3-4):315–347, 1996.

[49] Chris Sangwin. *Computer Aided Assessment of Mathematics.* Oxford University Press, 2013.

[50] Valerie J. Shute. Focus on formative feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[51] Valerie J. Shute and Joseph Psotka. Intelligent tutoring systems: Past, present and future. In D. Jonassen, editor, *Handbook of Research on Educational Communications and Technology.* Scholastic Publications, 1996.

[52] Sergey A. Sosnovsky, Michael Dietrich, Eric Andrès, George Goguadze, and Stefan Winterstein. Math-Bridge: Adaptive platform for multilingual mathematics courses. In Andrew Ravenscroft, Stefanie N. Lindstaedt, Carlos Delgado Kloos, and Davinia Hernández Leo, editors, *21st Century Learning for 21st Century Skills*, volume 7563 of *Lecture Notes in Computer Science*, pages 495–500. Springer-Verlag, 2012.

[53] Tarja Susi, Mikael Johannesson, and Per Backlund. Serious games – an overview. Technical Report HS-IKI-TR-07-001, University of Skövde, 2007.

[54] Kurt VanLehn. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, 2006.

[55] Kurt VanLehn, Collin Lynch, Kay Schulze, Joel A. Shapiro, Robert Shelby, Linwood Taylor, Don Treacy, Anders Weinstein, and Mary Wintersgill. The Andes physics tutoring system: Lessons learned. *International Journal on Artificial Intelligence in Education*, 15:147–204, 2005.

[56] Claus Zinn. Supporting tutorial feedback to student help requests and errors in symbolic differentiation. In M. Ikeda, K. Ashley, and T.-W. Chan, editors, *ITS 2006*, volume 4053 of *Lecture Notes in Computer Science*, pages 349–359. Springer-Verlag, 2006.

[57] Claus Zinn. Algorithmic debugging to support cognitive diagnosis in tutoring systems. In *Proceedings of the 34th Annual German conference on Advances in artificial intelligence*, KI'11, pages 357–368. Springer-Verlag, 2011.

Appendix A. Interactions with a domain reasoner

We illustrate how a learning environment and a domain reasoner interact by showing a typical sequence of request-reply pairs that are represented in XML. Suppose we want to practice solving a quadratic equation of medium difficulty in some learning environment. We start by requesting a new exercise by using the *generate* feedback service. In this example, terms will be encoded as strings.

```
<request service="generate" exerciseid="math.quadreq"
  difficulty="medium" encoding="string"/>
```

From the domain reasoner, we get back the equation $2(x^2 - 3) = 12$.

```
<reply result="ok" version="1.1 (5900)">
  <state>
    <prefix>[]</prefix>
    <expr>2*(x^2-3) == 12</expr>
  </state>
</reply>
```

The prefix that is part of the state corresponds to the position in the rewrite strategy. The state will be used in the next request to the domain reasoner. Suppose that we want to rewrite the equation into $x^2 - 3 = 6$ (dividing both sides by 2). To check this step, both the old state and the new equation are submitted to the *diagnose* service.

```
<request service="diagnose" exerciseid="math.quadreq" encoding="string">
  <state>
    <prefix>[]</prefix>
    <expr>2*(x^2-3) == 12</expr>
  </state>
  <expr>x^2-3 == 6</expr>
</request>

<reply result="ok" version="1.1 (5900)">
  <expected ready="false" ruleid="algebra.equations.coverup.times">
    <state>
      <prefix>[13,0,1,1,0,0,1,1,1,0]</prefix>
      <expr>x^2-3 == 6</expr>
    </state>
  </expected>
</reply>
```

The reply message indicates that the step is correct, and that the step was expected by the rewrite strategy. The reply gives us an abstract representation of the rule that was used (the identifier `coverup.times`) and returns a new state, which is not yet finished (the `ready` attribute). The state is used for the next request, where we rewrite the equation into $x^2 = 3$ and diagnose this step.

```
<request service="diagnose" exerciseid="math.quadreq" encoding="string">
  <state>
    <prefix>[13,0,1,1,0,0,1,1,1,0]</prefix>
    <expr>x^2-3 == 6</expr>
  </state>
  <expr>x^2 == 3</expr>
</request>

<reply result="ok" version="1.1 (5900)">
  <buggy ruleid="algebra.equations.buggy.plus"/>
</reply>
```

This step is not correct, as witnessed by the buggy-tag in the reply.[2] The buggy rule `algebra.equations.buggy.plus` corresponds to the common error of adding a term to one side of an equation, but subtracting the term on the other side. Based on this identifier, the learning environment can present a detailed description of the mistake to the learner. If this description is not sufficient for the learner to repair the mistake, all possible next steps can be requested with the *allfirsts* service.

```
<request service="allfirsts" exerciseid="math.quadreq" encoding="string">
   <state>
      <prefix>[13,0,1,1,0,0,1,1,1,0]</prefix>
      <expr>x^2-3 == 6</expr>
   </state>
</request>

<reply result="ok" version="1.1 (5900)">
   <list>
      <elem ruleid="algebra.equations.coverup.onevar.minus-left" location="[]">
         <state>
            <prefix>[25,0,1,1,0,0,1,1,1,0,0,1,1,0,0,1,0]</prefix>
            <expr>x^2 == 9</expr>
         </state>
      </elem>
   </list>
</reply>
```

The service returns one possible way to continue, namely by rewriting the equation into $x^2 = 9$ with a so-called minus-left cover-up rule. Instead of requesting a step, we could also ask for a worked-out solution. We use the service *solution*, which is called *derivation* in the current implementation.

```
<request service="derivation" exerciseid="math.quadreq" encoding="string">
   <state>
      <prefix>[13,0,1,1,0,0,1,1,1,0]</prefix>
      <expr>x^2-3 == 6</expr>
   </state>
</request>

<reply result="ok" version="1.1 (5900)">
   <list>
      <elem ruleid="algebra.equations.coverup.onevar.minus-left">
         <expr>x^2 == 9</expr>
      </elem>
      <elem ruleid="algebra.equations.coverup.power">
         <expr>x == 3 or x == -3</expr>
      </elem>
   </list>
</reply>
```

The worked-out solution in the reply shows the two steps that are needed to complete the exercise.

[2]The `result="ok"` attribute only means that the request was handled correctly.

request	*number*	*avg*	σ	Q_1	Q_2	Q_3
linear equations						
– diagnose	27,005	20.2	159.5	11.4	13.6	16.7
– solution	3,504	22.0	47.0	9.1	12.2	15.4
– onefirst	2,617	34.1	200.2	9.9	12.0	15.5
– allfirsts	1,332	19.2	68.1	8.9	10.9	12.9
– rulelist	574	21.5	59.1	6.0	7.2	8.1
– findbuggyrules	382	20.0	33.4	10.5	14.5	21.5
quadratic equations						
– diagnose	44,432	31.1	210.9	14.8	18.4	26.6
– onefirst	7,028	30.2	50.4	12.4	16.3	39.6
– solution	3,705	30.6	47.9	12.0	15.5	21.9
– allfirsts	1,590	17.1	28.3	10.3	12.6	15.6
– rulelist	801	17.1	64.7	6.7	9.0	9.9
– rulesinfo	92	1,291.6	915.9	228.2	1,662.0	1,879.2
– examples	66	63.5	43.1	41.3	47.0	59.0
higher-degree equations						
– diagnose	13,208	48.9	37.9	32.4	41.2	52.7
– onefirst	4,652	27.6	85.1	16.4	22.7	26.5
– solution	808	56.1	56.6	20.6	40.4	78.0
– rulelist	393	15.9	50.3	8.6	9.2	10.0
– allfirsts	155	22.8	41.2	11.2	13.7	17.4

Table B.7: Response times of services for polynomial equations in DME (grouped by exercise). Average (avg), standard deviation (σ), and quartiles (Q_1, Q_2, and Q_3) are given in milliseconds.

Appendix B. Response times of services

Table B.7 presents information about the distribution of the response times of services for requests from the DME. The data is grouped by exercise class, and was collected between April 2009 and October 2013. The numbers show that there is a considerable amount of variation in the measured response times: in many cases the average response time is greater than the third quartile, which indicates that there are a number of outliers in the data set. A more controlled experiment over a shorter period with a selected group of students is needed for measuring precise response times. Nevertheless, we feel that the data included in this paper provides an impression of the performance characteristics of our services. The anonymized data sets of response times for requests from the DME and the logic tool are available and can be downloaded from the authors' homepages.