A Perspective on Automated Assessment

Duarte P., Nunes I., Neto J.P., Chambel T ., Santos C.P.

Abstract

Which features should automated assessment systems have? We try to answer this question, looking at its implications in software design and development.

1 Introduction

The acronym CAA stands for Computer Aided Assessment, or some times Computer Assisted Assessment. A definition taken from the internet says "Computer Aided Assessment refers to the use of computers to assess students". This concept covers a very broad class of systems, and it is applicable in principle to all knowledge domains. We restrict the knowledge domain to Mathematics, and give the word "*aided*" the meaning of "*automated*", that is, the interaction during problem resolution is exclusively between the student and the computer, without any other human interference. The construction and usage of problem databases is implicit to all CAA(s). In the context of this discussion, a CAA system is understood as a tool for both assessment and problem authoring, where assessment comprises multiple kinds: summative, formative, and diagnostic or profiling.

This talk is a perspective on Automated Assessment that grew out of CATS (Computer Assessable Task System), a project that was motivated by the absence in existing CAA software of some features that we consider to be essential in any CAA system. We discuss here the most important ones.

In CATS we devised a model of an interactive computer assessment system that will allow:

- Students to solve elaborated mathematical problems, while being given feedback on their mistakes.
- Teachers to easily build new complex interactive problems from a pool of pre-defined modules.

The aim of CATS is the construction of a task-based computer assessment system, that proposes problems to students, and then guides, assesses, and provides feedback, throughout the resolution process. The model we devised for CATS is general enough to allow both teachers and students to deal with basic as well as complex tasks while keeping a simple structure at concept level.

Here, instead of describing CATS model, we shall discuss the features we think Automated Assessment systems should have.

2 FROM PROBLEMS TO TASKS

Any problem consists of a *statement* plus a *solution*, where by 'solution' we mean the formal specification of all acceptable answers, or resolution paths, through the definition of some *evaluating function*. This evaluating function not only decides whenever some student reply is to be accepted or rejected, but it may also provide explanatory feedback on wrong answers.

In the construction of problem databases it is important to abstract problems making them parametric. Given a particular problem, we find which data can vary without any loss of solution coherence. These data is then encapsulated in a few parameters, called *problem parameters*, which in general must satisfy some non trivial relations among them. We call these relations the *logical constraints* of the problem, and call *problem model space* to the set of all parameter data satisfying them. A parametric problem consists of a model space, a statement, and a parameter dependent solution, i.e., a parameter dependent evaluating function.

The art of creating nice parametric problems lies in the choice of simple, but sometimes sophisticated, problem model spaces. We say that a problem is *reverse engineered* when its parameter values are worked out from a sought parametric solution. In these cases, the solution can be retrieved from the problem parameters with zero computational cost. This means the evaluating function becomes trivial in what concerns the answer acceptance decision, however complex it may be in terms of the feedback it provides. We want to emphasize here that the computational complexity of an evaluating function will come from the pattern error recognition problem, much more than from the complexity of an algorithm to perform the student task automatically.

Finally, we need an algorithm to generate problem instances of a given parametric problem. Such algorithm works by randomly picking instances out of the corresponding problem model space. Sometimes we add to the logical constraints some *didactic constraints*, just to keep the difficulty level of problems homoge-

neous. The resulting algorithm is called a *model generating algorithm* for the parametric problem.

Notice that by keeping the concepts of *parametric problem*, and of *model generating algorithm* separated we allow parametric problems to be reused in different situations.

The concept of *task* generalizes that of *problem* and *parametric problem*. A task can, more generally, represent some general *type* or *class* of parametric problems. Each task consists of:

- a context,
- a set of typified parameters,
- an answer type, the type of answers that students must give,
- an evaluating function for student replies, which may produce explanatory feedback on wrong answers, and
- a, possibly empty, set of subtasks.

Tasks can be recombined, and reused, in arbitrarily complex tasks on account of the following two task hierarchies. The first is based on a relation *"being more general than"*, which either means that a task has more parameters than another, or else that the first task has more general parameter types than the second. This hierarchy is clearly fundamental for code reuse. A task re-usability relates well with its genericity. Specific problems are hardly re-usable, while generic ones can be extensively reused. The second hierarchy is based on the relation *"being composed of"*, meaning containing another task as a subtask. A task is a kind of function whose arguments are its parameters, and whose returned value is the answer achieved through student-machine interaction. Subtasks may communicate with each other in some way specified by the task, meaning that some subtask answer may be used as another subtask parameter. Contexts work as protocol languages allowing communication between different subtasks of a given task. This hierarchy is key to achieve problem complexity. The idea is that generic tasks, made by experts, will be used as building blocks by less skilled teachers when assembling their problems.

3 TASK EXAMPLES

In this section we describe a list of task examples.

3.1 Equation Solvers

In [10] the authors describe an interactive equation solver, whose kernel is implemented in Haskell, a generic functional programming language. We like to regard this application as a realization of the Equation Solver task idealized here.

`OneStepEquationSolver` is a simple task, that allows the user to progress one step in the resolution of an equation. It is not designed as a stand-alone task, but rather to be used as a sub-task of `EquationSolver`. Its evaluating function should recognize error patterns in algebraic manipulation, providing feedback on mistakes.

Context:	Algebraic equations
Parameter:	An equation
Answer Type:	An equation
Evaluating Function:	Decides if two equations are equivalent.
Subtasks:	None

`EquationSolver` is a basic task that assesses students throughout the resolution of an algebraic equation. The subtask `OneStepEquationSolver` is called step-by-step, with the last accepted equation as parameter. In [10] the authors define *progress indicators* to monitor resolution progress in their equation solver. Based on this information, guidance can be provided to the student. Notice this task provides two layers of feedback, based on its own, and its sub-task, evaluating functions.

Context:	Algebraic equations
Parameter:	An equation
Answer Type:	An equation
Evaluating Function:	Evaluates progress in equation resolution, deciding when it is solved.
Subtasks:	`OneStepEquationSolver`

3.2 Provers

We call *Provers* to tasks designed to assist and assess students throughout mathematical proofs. These tasks use the *Natural Deduction Proof System* proposed

by the German mathematician G. Gentzen in 1935, which captures the natural way of thinking. Each prover context is either a simple logical context, or else a toy Mathematical formal theory, whose axioms are the principles and theorems of some elementary subject of Mathematics. General provers are tools that assist students to perform formally correct mathematical reasoning, providing feedback on logical fallacies. Provers keep the state of their *developing proofs* updated. Each deductive step is accomplished through a sub-task, `JustifiedStatement`, which checks if a stated conclusion can be inferred from the specified list of premises.

`JustifiedStatement` This task should support all Natural Deduction System demonstration schemes, like *hypothesis introduction*, *reductio ad absurdum*, *proof by cases*, etc. It admits as single parameter the on going proof of the invoking prover. This task evaluating function will figure out which demonstration scheme, or inference rule, to use in each deductive step. This is specially important for students not fully acquainted with Mathematical reasoning. The evaluating function will also recognize logical fallacies, allowing to explain to the student the nature of his mistakes. This evaluating function can also be "taught" to assume certain assumptions as implicit whenever needed. In order to support such feature we use *deductive filters*, which filter sublists of implicit premises from given proofs. This deductive filter can be seen as an extra parameter of the task `JustifiedStatement`.

Context:	Mathematical Formal Theory
Parameter:	A proof
Answer Type:	A stated conclusion, and a list of premises
Evaluating Function:	Checks whether the conclusion follows from the premises
Subtasks:	None

3.3 Geometric Sketchers

The tasks described here have long been reified in Dynamic Geometry Software such as *Cinderella* [4], *Cabri* [3], or *Geometer's Sketchpad* [11]. All the tasks bellow take a *geometric model* as its unique parameter. Let us precise what we mean by a *model* here. Given a logic context $\mathfrak{C}$, a model in $\mathfrak{C}$ consists of

- a list of variable names, representing objects of $\mathfrak{C}$,

- a list of statements, describing the model relations, or constraints, on the model variables,
- a sublist of names, standing for the model *free elements*.

We assume that every model is such that all non free element get uniquely determined from the free ones by the model relations. We call *model instance* to any assignment of values to the model variables, determined by a choice of values for the free variables. Thus, by definition, any instance of a model satisfies its list of assumptions. Models of geometric contexts will be referred to as geometric models.

`MakeConstruction` is a basic task allowing the user to further elaborate a given geometric model. Although the internal aspects of this task are not important, it can be built with multiple sub-tasks, each corresponding to a specific tool or action, such as to introduce new elements to the geometric model, or to re-name an existing element. These tools take a geometric model as their single parameter, and return the modified geometric model as their answer object. The task itself has no answer type. Its geometric model parameter simply gets updated by the sub-task actions.

Context:	Geometry
Parameter:	A model
Answer Type:	None
Evaluating Function:	None
Subtasks:	All available tools

`MakeStatement` is a task that allows students to state facts about geometric constructions (models), whose truth is semantically checked. This means that the statement is checked to satisfy some random sample of instances of the geometric construction (model). This is a probabilistic approach. When a statement is accepted as true, the chance of it being wrong is negligible, and if a statement does not satisfy some model instance, the underlying geometric construction can be exhibited as part of an explanation of the statement incorrectness. We note that *Cinderella* [4], software has a *theorem proving* mechanism that works just as this idealized task.

Context:	Geometry

Parameter:	A model
Answer Type:	A statement
Evaluating Function:	Checks the statement truth
Subtasks:	None

3.4 Geometric Provers

Geometric models contain lists of geometric assumptions. In task `MakeConstruction`, each step in the elaboration of its geometric model corresponds to the definition of a new variable in terms of previously introduced variables. Such a definition is a new statement added to the underlying model. The task `JustifiedStatement` can produce equation type statements, and an equation solved by task `EquationSolver` is again a geometric statement. Thus, we can mix these three tasks as subtasks of a powerful Geometric Prover which, instead of a list of assumptions, takes a geometric model parameter.

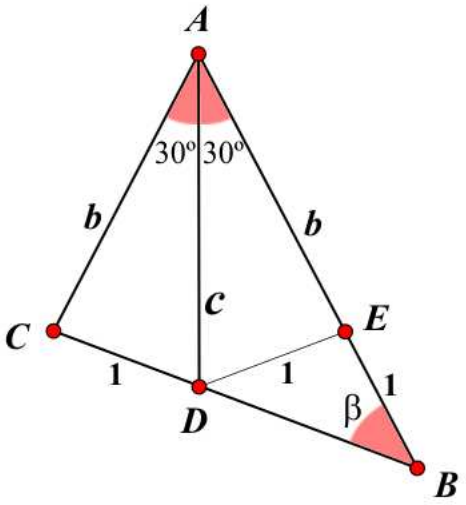
`GeometricProver` A prover for geometric problems

Context:	Geometry
Parameter:	A geometric model
Answer Type:	None
Evaluating Function:	None
Subtasks:	`EquationSolver`, `MakeConstruction`, `JustifiedStatement`

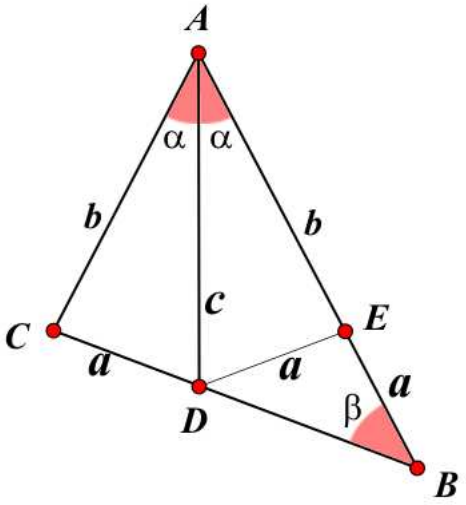
The subtask `JustifiedStatement` can be specialized to contain `MakeStatement` as its subtask, in such a way that syntactical deductive inference is preceded by semantical truth checking. This enhances task `JustifiedStatement` with an extra feedback layer.

3.5 A Geometric Problem

Consider a teacher problem of determining the angle $\beta = \angle CBA$, and the lengths b and c in the following geometric construction



We model this problem through the geometric constructions depicted in the following image



where:

- $E \in AB$, $D \in CB$,
- $a = |CD| = |DE| = |EB|$,
- $b = |AC| = |AE|$,
- $c = |AD|$,

- $\alpha = \angle CAD = \angle DAB$,
- $\beta = \angle CBA$.

This problem model space has dimension 2. It is a geometric model with free parameters α and a, that we shall denote by $\mathcal{M}(\alpha, a)$. Using this model as a parameter, the teacher can now easily specialize a Geometric Prover, to obtain a solver task for his problem.

`GeometricProblem` task that specializes `GeometricProver` with parameter $\mathcal{M}(\alpha, a)$.

Context:	Geometry
Parameters:	α and a
Answer Type:	None
Evaluating Function:	Checks whether the current proof contains goal statements of the form $\beta = E_\beta$, $a = E_a$, where E_β and E_a are numeric expressions.
Subtasks:	Same as `GeometricProver`

An outer layer of feedback can be added by the teacher to his `GeometricProblem` task, for instance specifying subgoals and setting the evaluating function check for their attainment, providing feedback accordingly. Another way would be to associate these subgoals with subtasks specializing `EquationSolver`, `MakeConstruction`, or `JustifiedStatement`, establishing resolution precedences among these subtasks, which would provide some level of guidance along the problem resolution.

4 CAA FEATURES

We end this presentation with a list of features that we believe CAA should have.

Re-Usability

It is obviously a strategically good idea to foresee and encourage code produced in constructing problem databases to be reused as much as possible. In software engineering code reuse is related to its genericity and modularity.

Genericity

A generic task is an abstract task, in the sense that it captures the main, common,

structure and characteristics of whole classes of problems, abstracting away all specific features of each possible subclass. Once defined, they allow re-usability of those common parts.

Modularity

Modularity is at the core of compositional systems of reusable *modules – tasks* in CATS – encapsulating their internal behaviour. Clearly, modularity is essential to achieve some degree of problem complexity. In a non modular system, each problem code becomes essentially a one author job. As a consequence, this type of system has author patience as a non trivial constraint on problem complexity. On the contrary, in a modular system, a problem module can be the product of many independent contributors. Therefore, there is no *a priori* bound on the problem complexity.

Feedback

Rich feedback is a fundamental feature for any CAA system. How can we achieve it? Computer Algebra Systems (CAS) are invaluable in what function evaluation is concerned, since they possess huge libraries of functions and algorithms.

Many existing CAA systems use some Computer Algebra System (CAS) to evaluate students answers: Maple TA [7] and Aim [1] use Maple [7], Stack [12] uses Maxima [8] or Axiom [2], LeActiveMath [6] and MathDox [5] use a variety of CAS and theorem provers, Wallis [13] uses CAS supporting MathML or OpenMath like Maple or Yacas [14].

However, CAS functions do not provide feedback. In non modular systems, feedback messages have to be hard-coded by the teacher in the process of creating an exercise. We advocate a different approach, where a problem can have several feedback layers, of which the last one alone is left to the teacher. Feedback information should be treated in a similar way to *exception systems* of state-of-art programming languages. In a modular CAA, its modules, the tasks, should have an encapsulated ability of sending feedback as error messages, when something goes wrong about the student answer. When a new problem or task is created as composition of several tasks, its author specifies how the subtasks' messages should be caught and treated before being sent to the student. The author may also add to them its own feedback messages, triggered by error conditions that he/she specifies. Evaluating functions capable of recognizing error patterns, and generating feedback messages, will be central in such a system of feedback messages. We do not consider CAS particularly advantageous to define them, when compared with other more flexible programming languages, specially because, in

many problems, a smart reverse engineered problem model dismisses the need for complex CAS algorithms.

Generality

A CAA should be general enough to allow the representation of other CAA problems, without loss of expressiveness.

Client Running System

The CAA system should be able to run both on the server side, for summative assessment, and on the client side, for formative assessment. This would avoid numerous inconveniences caused by internet traffic problems.

Multiple Kernels

A logic kernel is essential to any CAA for two reasons:

- to execute the model generating algorithm, which randomly generates different versions of a parametric problem,
- to evaluate student answers, through an evaluating function call, that generates feedback.

Many CAA use CAS as kernels. Some are based on a specific kernel, and others allow different CAS kernels. We believe it is a good idea to build a CAA system that supports multiple kernels, some of which may be general programming languages, instead of just CASs. The *generality* and *feedback* arguments above support this option. Two more arguments favor this choice, the *client-running system* option, and the *commercial* aspect of most popular CASs.

Community Work

A community of pleased users and developers should be the fastest and safest way for a CAA system to thrive. There are many examples of such communities dedicated to specific programming languages.

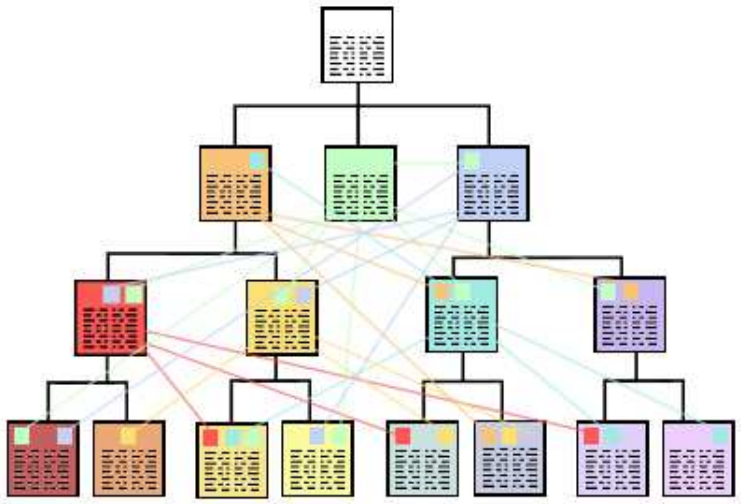
Open Source Model

This is a corollary of all previous options.

5 CAA Architectures

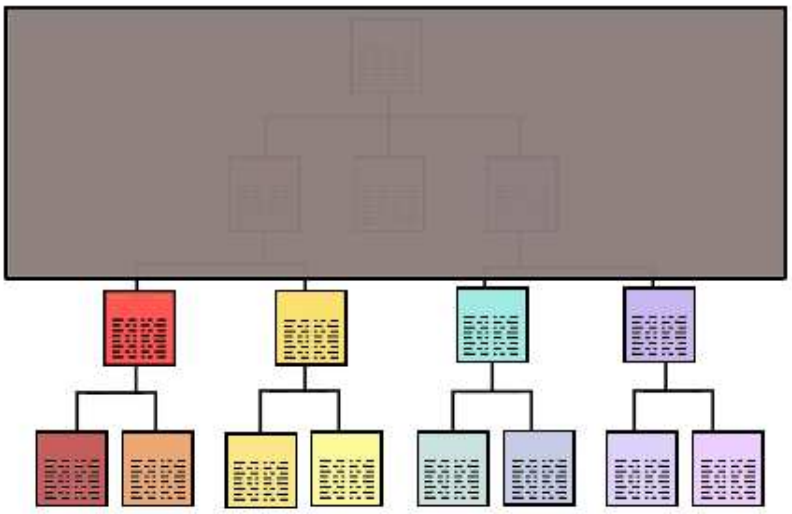
It would be natural to expect that every CAA is based on some internal *problem type hierarchy*, with the most general problem specification at the top, and problem instances at the bottom. We end this presentation with a comparison of CAA according to the unveiling of this hierarchy.

Open Type of Architecture



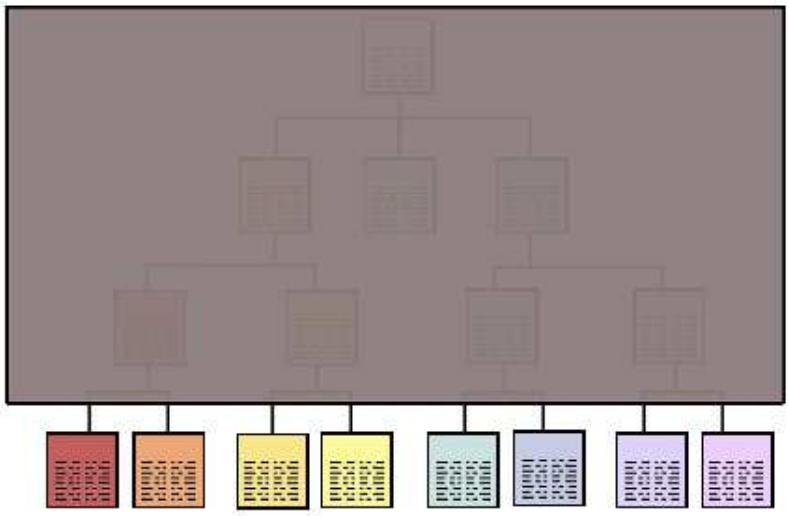
This type of CAA is characterized to leave the *problem type hierarchy* wide open, allowing easy creation of new generic problems at any level of this hierarchy. It is designed to take advantage of a community of users, both teachers and programmers, where the work of a few, not just the developers, can really be reused by many. CATS will eventually be a CAA system of this kind. CATS will be a modular system, another key feature in what code reuse is concerned, through another independent hierarchy: that of problem composition, where problems, or tasks, gain a new functional character.

Semi-Closed Type of Architecture



This is perhaps the most common type of current CAA systems. It is both an assessment and authoring tool. Problem authors, hereafter referred as *teachers*, have access to a more or less long list of *problem types* to make their problems. We believe that, in all CAA systems, these multiple problem templates are part of some problem type hierarchy, where they get unified in a single and most general problem type. For either commercial or practical reasons, such as easiness in problem authoring, most of this problem type hierarchy is kept hidden, or unaccessible, to problem authors. It is obvious that, for teachers with few skills in programming or computer technologies, it is rather easier to master a long list of ready to use problem templates, than to work with a single but flexible problem object which has to be tailored to meet his/her needs. But there is one handicap to this approach. Problems built with these type of CAA are final elements in the problem type hierarchy. Therefore the semi-closed architecture type is incompatible with genericity. Semi-closed systems are essentially non modular, although some allow for a weak form of composition. In these type of CAA, any complex problem, with multiple interdependent questions, is always a one author job. This means that, in such systems, code reuse is at best reduced to some sophisticated form of *copy-n-paste*.

Closed Type of Architecture



This kind of CAA is just an assessment tool, not an authoring tool. The whole system is a *black-box*, used by developers to produce problem databases. One such example is ALEKS. It is a commercial integrated learning system, with a strong assessment component that keeps a profile on each student *knowledge state*.

6 Conclusions

We motivated, described and discussed the features that we find essential in any Computer Aided Assessment system. At the core of any CAA system should be an open and modular architecture in order to enhance re-usability in the task creation process and, thus, promoting easiness in problem definition, richer problem collections, and more and better feedback to students.

In the context of the CATS project [9] we investigate the issues involved in the prosecution of those objectives, that is, a CAA system possessing the properties we deemed essential.

References

[1] Aim. http://maths.york.ac.uk/moodle/aiminfo/.

[2] Axiom. http://page.axiom-developer.org/.

[3] Cabri. http://www.cabri.com/.

[4] Cinderella. http://cinderella.de/.

[5] Mathdox. http://www.riaca.win.tue.nl/.

[6] Leactivemath. http://www.leactivemath.org/.

[7] Maple ta. http://www.maplesoft.com/.

[8] Maxima. http://www.maxima.sourceforge.org/.

[9] Duarte P., Nunes I., Neto J.P., Chambel T., and Santos C.P. Enhancing modularity and feedback in computer aided assessment. In *Proc. 15th International Conference on Computing (CIC 2006)*, pages 240–246. IEEE Computer Society, 2006.

[10] Harrie Passier and Johan Jeuring. Feedback in an interactive equation solver. In *Proceedings of the Web Advanced Learning Conference and Exhibition, (WebALT 2006)*, 2006.

[11] Geometer's sketchpad. http://www.dynamicgeometry.com/.

[12] Stack. http://eee595.bham.ac.uk/ stack/.

[13] Wallis. http://www.maths.ed.ac.uk/ wallis/.

[14] Yacas. http://www.xs4all.nl/ apinkus/yacas.html.