STUDENT MODELLING BY ADAPTIVE TESTING - A KNOWLEDGE-BASED APPROACH

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Sophiana Chua Abdullah

June 2003

To my family,

Abstract

An adaptive test is one in which the number of test items and the order in which the items are presented are computed during the delivery of the test so as to obtain an accurate estimate of a student's knowledge, with a minimum number of test items. This thesis is concerned with the design and development of computerised adaptive tests for use within educational settings. Just as, in the same setting, intelligent tutoring systems are designed to emulate human tutors, adaptive testing systems can be designed to mimic effective informal examiners. The thesis focuses on the role of adaptive testing in student modelling, and demonstrates the practicality of constructing such tests using expert emulation.

The thesis makes the case that, for small scale adaptive tests, a construction process based on the knowledge acquisition technique of expert systems is practical and economical. Several experiments in knowledge acquisition for the construction of an adaptive test are described, in particular, experiments to elicit information for the domain knowledge, the student model and the problem progression strategy. It shows how a description of a particular problem domain may be captured using traditional techniques that are supported by software developed in the constraint logic extension to Prolog. It also discusses knowledge acquisition techniques for determining the sequence in which questions should be asked.

A student modelling architecture called SKATE is presented. This incorporates an adaptive testing strategy called XP, which was elicited from a human expert. The strategy, XP, is evaluated using simulations of students. This approach to evaluation facilitates comparisons between approaches to testing and is potentially useful in tuning adaptive tests.

Acknowledgements

To the many people ‘behind the scenes’, I thank you all:

To my supervisor, Dr Roger E. Cooley, for his unwavering and unstinting guidance throughout the PhD course.

To the friends I have made over the years, who in their different ways, have made my stay away from home a most memorable one, in particular, Sukaina, Florence, Pamela C., Claudio, Francisco, Steve, Tom, Andy, Edries, Chris, Justine, Kerry, Sinan, Huda, Haytham, Birgit, Karen G., Peter, Karen D., Freddie, Ros, Jane, Joseph, Simone, Wuen Hao, Chandra and Pamela.

To my family who has been a tower of strength, in particular my parents, Dato and Datin Chua Kong Leng, and my parents-in-law, Pengiran Hassan and Peni Jurof. To my husband, Pengiran Abdul Wahab, and my children, Ak. Mohammad Hanif, Dk. Hawa Hafizah, Ak. Mohammad Hazim and Dk. Hawa Hayati, who have all been there for me. To my sisters, sisters-in-law, and Zenaida for playing the mother figure to my children during the months of my absence.

To His Majesty’s Government of Brunei, the Ministry of Education and Institut Teknologi Brunei, my deepest gratitude for the financial support and the opportunity to perform research.

Table of Contents

List of Figures

List of Tables

Chapter 1.

Introduction

In the context of education, a “test” is usually a series of questions. Typically the sequence of questions is fixed, but in an adaptive test, the selection of questions is partially determined by the responses to earlier questions in the sequence. This thesis is concerned with the design and development of adaptive tests in the context of student modelling within intelligent tutoring systems. It proposes a strategy for constructing such tests based on expert emulation.

This chapter presents the motivation and aim of the research and an outline of the thesis.

1.1 Motivation

The last three decades have seen a considerable effort to develop intelligent tutoring systems which provide tuition that is tailored to the needs of individual students. The individualised attention such systems offer is made possible through *student modelling*.

Student modelling is concerned with the task of keeping a record of many aspects of a student (Greer and McCalla, 1991). Such a record is called a student model (Self, 1974) and it may include *domain-specific information,* such as how much and what the student has learned to date, what misconceptions he or she may have, and what problem solving strategies he or she may possess, and *learner-specific characteristics*, such as what learning styles seem to be successful for the student, and what conative and affective dimensions would impact the student. The task of inferring such information from learner's behaviour is a major challenge in student modelling.

In recent years, computerised adaptive testing has gained popularity as a student modelling technique (for example, (Huang, 1996) (Collins *et al.*, 1996) (Dowling and Kaluscha, 1995) (Ríos *et al.*, 1999)). Though originally used for tests of competence rather than for diagnostic purposes, computerised adaptive testing is useful when deep cognitive modelling is not necessary. This is the case when the aim of an intelligent tutoring system is to present remedial teaching based on an assessment of the student's domain-specific knowledge in terms of what he or she knows. When attempting to infer a student's knowledge in terms of aspects such as his or her problem solving strategies and misconceptions, adaptive testing is not appropriate.

Computerised adaptive testing is characterised by the use of the minimum number of questions of 'appropriate' difficulty, in order to determine, with high accuracy, the level of performance of the student (Welch and Frick, 1993). It is superior to the conventional fixed-item pencil-and-paper tests in that it has the effect of reducing test anxiety and the overall testing time. Though computerised adaptive tests can be both accurate and efficient, they are not necessarily any easier to construct than other student modelling programs. A review of literature reveals two major approaches of computerised adaptive testing commonly used in student modelling. They are the Item Response Theory (Wainer and Mislevy, 1990) and Knowledge Space Theory (Falmagne *et al.*, 1990). The first approach uses a statistical model and requires large empirical studies to calibrate its questions against student populations. This is not feasible for small-scale construction of adaptive tests for use within classrooms. Also, such tests perform summative assessment and represent a student's knowledge of a subject domain as a single proficiency estimate only. The second approach does not require large empirical studies and has adopted expert emulation in eliciting the problem progression strategy of adaptive testing. The result of diagnosis takes the form of a 'knowledge state' which represents the set of problems or skills that the student has displayed mastery of. This is more closely related to the diagnostic endeavour of many student modelling systems.

The thesis is mainly concerned with tutoring in an environment in which the knowledge state of a student is the most significant determiner of the form of remedial tutoring. The motivation behind the research is to examine the feasibility of eliciting the strategies of a human teacher or tutor for the *whole* process of adaptive testing, from the construction and representation of the test syllabus to the problem progression or testing strategy itself. While expert emulation is common in designing many intelligent tutoring systems (Seidel and Park, 1994), it is not a common practice in the design of adaptive tests. Adaptive testing systems can be designed to emulate an effective informal examiner. As Wainer (1990) pointed out, the basic notion of an adaptive test is "to mimic what a wise [human] examiner would do".

Adaptability is the key attribute of human intelligence (Boy, 1996) and it is this trait which an effective human tutor possesses that enables him or her to provide one-on-one tutoring and adapt to the needs of the individually different student. This is what Philip of Macedon's son, Alexander, had enjoyed as a royal prerogative: the personal services of a tutor as well-informed and responsive as Aristotle (Suppes, 1966). Studies have shown that the knowledge of human tutors is rich and varied (Putnam, 1987) and that human tutoring provides the most effective method of instruction (Bloom, 1984). Since the study by Bloom revealed that one-on-one human tutoring is the most successful form of instruction with 2-sigma learning gains over classroom teaching, designers of intelligent tutoring systems have replicated this finding with computer tutors where computers generate adaptive forms of tutoring for individual learners (du Boulay, 2000a). This means that by studying human tutors, intelligent tutoring systems can be designed and developed to provide individualised or adaptive instruction (Park, 1996) and to act much like a private tutor by aiming at 2-sigma learning gains.

The hope underpinning this thesis is that just as human one-on-one tutoring has been proven to be the most effective form of instruction, human one-on-one testing to assess the state of knowledge of a student is the most effective form of assessment, and that the testing strategy is worth capturing. The thesis makes the case that for small scale adaptive tests, a construction process based on the knowledge acquisition technique of expert systems is practical and economical. The end result of such emulation is a student modelling tool for adaptive testing which can be used by human teachers themselves.

1.2 Aim of Research

The aim of the research is to develop a strategy for the design and construction of small-scale adaptive tests based on expert emulation. In one-on-one tutoring, there are three main types of knowledge which a human tutor needs to achieve effective tutoring (Self, 1974) and these knowledge components form the building blocks of an intelligent tutoring system (Ohlsson, 1987). Similarly, in one-on-one testing, there are three main types of knowledge which an informal examiner needs to maintain in order to render effective assessment. This is the knowledge about *what* to test (the domain knowledge or testing syllabus), *who* to test (the student model) and *how* to test (the problem progression strategy). The types of knowledge so gained from a human teacher or tutor can be represented in a student modelling architecture called SKATE shown in Figure 1.

SKATE, which stands for **S**tudent **K**nowledge assessment by **A**daptive **T**esting and **E**xpert emulation, represents the adaptive testing strategy of the expert. Its function is to model the knowledge of the student in a subject domain by adaptive testing and to build a student model which can be used to guide subsequent remedial help. In Figure 1, the student modeller orchestrates the test administration while the interface module facilitates communication between SKATE and the human student.

The following questions need to be answered:

- *Is there an efficient and effective approach of capturing such information?*
- *Is there an efficient way of representing such information?*

In order to answer these questions, the following tasks were identified:

a. to review current student modelling and adaptive testing techniques,
b. to survey the conventional knowledge acquisition techniques,
c. to investigate the potential use of constraint logic programming as a knowledge acquisition tool,
d. to carry out expert emulation based on the techniques identified in b. and c. above,
e. to incorporate the information and strategies gained from expert emulation in a student modelling architecture, and,
f. to evaluate the student modelling strategy.

These tasks are handled by different parts of the thesis. Task a. is examined in Chapter 2. Tasks b. and c. are investigated in Chapter 3. Task d. is carried out in Chapters 3 and 4 while task e. is handled by Chapter 5. Task f. is carried out in Chapter 6.

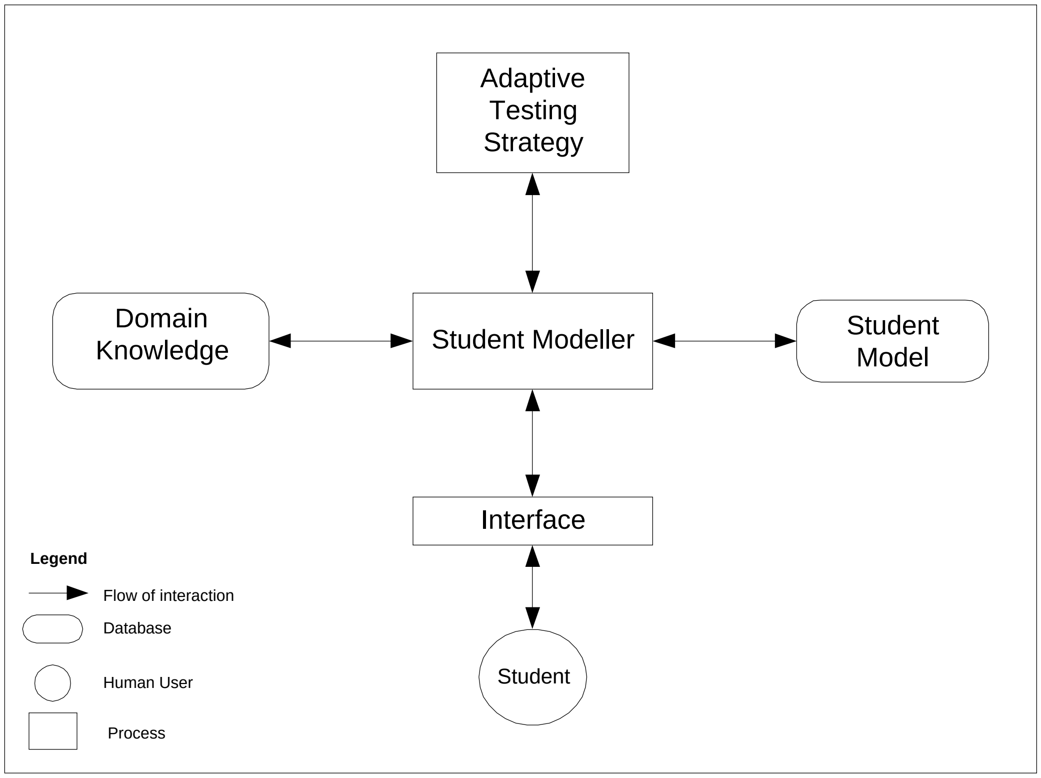

Figure 1. The Architecture of SKATE

1.3 Outline of Thesis

This thesis consists of seven chapters including this introductory one. An overview of the subsequent six chapters now follows.

Chapter 2 presents a review of literature on student modelling in intelligent tutoring systems and focuses on computerised adaptive testing as a recent advent in student modelling. It first describes the intelligent tutoring paradigm and examines the major student modelling techniques which model domain-specific knowledge of the student. Challenges associated with student modelling and uncertainties are addressed. The chapter further discusses computerised adaptive testing in the light of other testing procedures such as fixed-item testing and self-adaptive testing. Two major approaches of computerised adaptive testing which have been influential in student modelling are described along two aspects – the construction of the domain and the problem progression strategy.

Chapter 3 first lays the foundation for knowledge acquisition by describing the context in which the research is conducted. It includes a description of a small experiment called MATT which confirms that the expert teacher performs adaptive testing when assessing a student's knowledge in a subject domain on a one-on-one interaction. Knowledge acquisition techniques are discussed including the potential use of constraint logic programming in numeric domains. The chapter then discusses the results of knowledge elicitation in the construction of the domain knowledge which represents the test syllabus. Other issues like identifying problem solving skills, generating problems, diagnosing student answers and measuring problem difficulty are discussed. It also shows how Constraint Logic Programming can be used for knowledge elicitation, knowledge representation, problem generation and answer evaluation.

Chapter 4 discusses the experiments in creating a student model and a problem progression strategy in SKATE. It first discusses the usefulness of a student model for adaptive testing and seeks to determine the contents of such a model. It bases its decision on the findings of two experiments, such as the remediation strategy of the expert after testing and concludes with the choice to maintain domain-specific information about the student, in terms of what is believed to be mastered and a record of successful and unsuccessful attempts at problems. A clp(FD) representation means that the overlay student model is executable and is useful for predicting a student's performance and for generating problems during remediation. The chapter next presents two distinct strategies for problem progression based on expert emulation. The first experiment is the development of a computer-aided procedure to systematically query an expert to extract a test item sequence called BT. The second experiment describes a knowledge elicitation exercise which captures the expert's testing strategy called XP, which is based on his measure of problem difficulty by the number of skills needed to solve a problem.

Chapter 5 describes the SKATE architecture and its knowledge components. It is the culmination of the elicitation work carried out in the previous three chapters. Fraction addition is the example domain used throughout the study but the proposed architecture should be able to support other topics in mathematics. The student modelling component of SKATE called XP is described.

Chapter 6 describes an evaluation strategy carried out to measure the performance of XP student modelling strategy. Since the circumstances under which this work was undertaken precluded classroom testing, simulations of student behaviour are used. The results from the evaluation are compared with those of a sequential testing strategy, ST. Different versions of XP are created and evaluated against XP and ST.

Chapter 7 provides a summary and findings of the work achieved so far. It presents the main contributions of the research and concludes with directions for future work.

1.4 Miscellaneous

The masculine will be used throughout the manuscript to denote both male and female. The following acronyms are used in the thesis:

• CAI	Computer-Assisted Instructional
• ICAI	Intelligent Computer-Assisted Instructional
• ITSs	Intelligent Tutoring Systems
• CAT	Computerised Adaptive Testing
• IRT	Item Response Theory
• KST	Knowledge Space Theory
• clp(FD)	Constraint Logic Programming over Finite Domains
• FIT	Fixed-Item Testing
• SAT	Self-Adaptive Test

In order to avoid repetition, there will be constant referencing to different parts of the thesis. For example, the phrase “see Section 2.3” means see Section 2.3 of Chapter 2, where the integer part denotes the chapter number.

Chapter 2.

Student Modelling, Intelligent Tutoring and Adaptive Testing

2.1 Introduction

This chapter presents a review of literature on student modelling in intelligent tutoring systems and focuses on computerised adaptive testing as a recent advent in student modelling. It first describes the intelligent tutoring paradigm and examines the major student modelling techniques which model domain-specific knowledge of the student. Challenges associated with student modelling and uncertainties are addressed. The chapter further discusses computerised adaptive testing in the light of other testing procedures such as fixed-item testing and self-adaptive testing. Two major approaches of computerised adaptive testing which have been influential in student modelling are described along two aspects – the construction of the domain and the problem progression strategy. The chapter concludes with a discussion on the implications of the review of these two techniques on the design of SKATE.

2.2 Intelligent Tutoring Systems

Computers have been used in education for more than three decades (Mandl and Lesgold, 1988). With lowering costs and continual improvements in computer technology, computers are becoming more affordable and accessible to educational institutions. The widespread use of computers in classrooms has been mainly in information processing software applications such as word processors, spreadsheets and database systems, and to a certain degree in subject-specific computer-assisted instructional or CAI systems.

The CAI systems of the 1950s were influenced by the behaviourist psychological theories (Skinner, 1954) and were simple *linear* programs. These evolved into *branching* programs in the 1960s which offered a degree of adaptability to the student. An action of the student may correspond to a branch point in the program and a branch is selected based on the response of the student. There may be a script for each branch point, for example, "if question 2 answered correctly, go to question 10, else go to question 3". Branching programs evolved into *generative* programs in the 1970s (Uhr, 1969). These are frame-oriented tutoring systems and could generate new problems from a combination of different elements in a database but still, adaptability was limited and often unrelated to the individual student needs. The instructional design was hard-wired into the domain content material as simple branches and loops. There was no inferencing about the student's learning state. These early systems could not handle complex student responses and did not explicitly address the issues of how students learn. The assumption was that if these systems presented information to the learner, the learner would absorb it (Urban-Lurain, 1996). Although there have been notable successes, the architecture of CAI systems has been inadequate to provide robust and rich learning environments (Clancey and Soloway, 1990).

As computer technology became more sophisticated in terms of processing power, storage facilities, peripheral designs, graphical user interfaces and networking, researchers began to think about CAI systems which offered more *individualised* attention to students. They began to look at other techniques, such as Artificial Intelligence, in the attempt to produce effective computer tutors which emulate a good private human tutor who "has the ability to perceive a student's view and adapt their behaviour accordingly" (Wenger, 1987). This prompted research in Intelligent Computer-assisted Instructional or ICAI systems in the mid 1970s. This interest coincided with the emergence of expert systems which use Artificial Intelligence techniques to mimic human experts in fields such as medicine and engineering. Along the same vein, ICAI systems were developed with the aim to mimic human tutors.

According to the handbook of Artificial Intelligence, ICAI systems represent one of the main applied fields of Artificial Intelligence (Barr and Feigenbaum, 1981). In 1982, the field acquired its most popular name, Intelligent Tutoring Systems (or ITSs), when Sleeman and Brown (1979) published a special issue of the International Journal of Man-Machine Studies as a book (Sleeman and Brown, 1982). Early examples of ITSs are compiled by Barr and Feigenbaum (1981) and Polson and Richardson (1988) and include SCHOLAR (Carbonell, 1970), GUIDON (Clancey, 1979) and DEBUGGY (Brown and Burton, 1978). Many other textbooks, surveys and reviews have been published by key researchers such as Self (1974, 1979, 1988a, 1988b, 1995, 1999a, 1999b), Wenger (1987), Clancey and Soloway (1990), Anderson (1992) and Shute and Psotka (1996).

The intelligent tutoring paradigm draws its characteristics and strengths from many different disciplines. The development of such programs lies at the intersection of Computer Science, Cognitive Psychology and Educational Research; this field is often referred to as *Cognitive Science* (Kearsley, 1987) – see Figure 2.

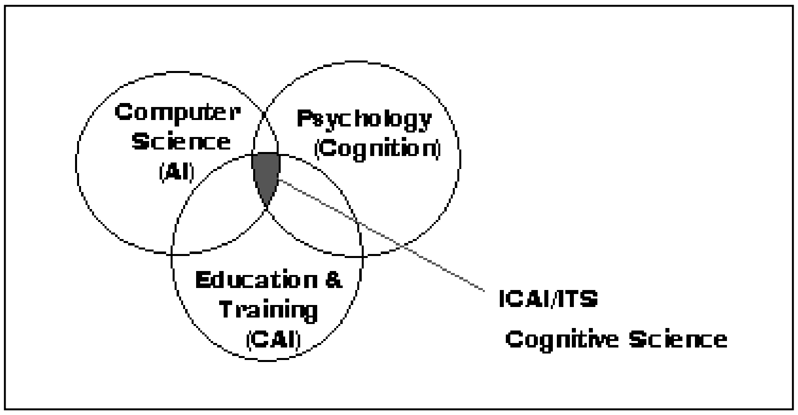

Figure 2. ICAI Domains (Kearsley, 1987)

From the field of education and training, ideas on teaching and learning strategies were adopted such as one-on-one tutoring, collaborative learning, peer-to-peer interaction and learning companions. From the field of Artificial Intelligence and Expert Systems, techniques such as machine learning, fuzzy logic, rule-based inferencing, and Bayesian network inferencing featured in many intelligent tutoring and student modelling endeavours. Ideas were also drawn from the field of cognitive psychology. This includes the work of John Anderson who developed the ACT theory of cognition (Anderson, 1983) and built several computer tutors using this theory. Examples include the LISP Tutor (Anderson and Reiser, 1985) and the Geometry Tutor (Anderson *et al.*, 1985). The ACT theory has evolved as ACT*, ACT-R and more recently as ACT-R/PM which has the potential of modelling high-density sensing information such as the tracking of eye movement and speech recognition (Anderson, 1998b).

The field of intelligent tutoring is relatively young. Since its inception nearly three decades ago, interest was at its prime in the late 1970s and 1980s. However, interest began to wane when difficulties in developing effective ITSs, especially in student modelling, were encountered. While a few ITSs have been deployed into real life settings, many still remained within research laboratories. Today, the field of ITS is in a state of equilibrium and is still actively researched, as is evident in the proliferation of current journals and conferences:

- Journal of Artificial Intelligence in Education (IJAIED)
- International Conference on Intelligent Tutoring Systems (ITS)
- International Conference on Artificial Intelligence in Education (AIED)
- International Conference on User Modeling (UM)
- World Conference on Computers in Education (WCCE)
- International Conference on Computers in Education and International Conference on Computer Assisted Instruction (ICCE/ICCAI)

Today, ITS research spans across many subfields such as:

- student modelling
- agent-based tutoring systems
- authoring tools
- ontological engineering
- ITS architectures
- distributed learning environments
- instructional design
- web-based education

Today, the aim of many ITS designers is to design and build computer tutors which can offer their students *nearly the same degree of reasoning* as achieved with a human tutor. As mentioned in Section 1.1, a study conducted by Bloom (1984) revealed that one-on-one human tutoring is the most successful form of instruction. Bloom found that one-on-one human tutoring shifted the distribution of achievement scores of students by about two standard deviations or two sigma learning gains compared with the conventional classroom teaching. This is shown in Figure 3 where the class average moves from 50th percentile to the 98th percentile. This two standard deviation improvement, or Two Sigma shift, has become a goal at which designers of ITSs aim (du Boulay, 2000b). To date, no intelligent tutoring system has attained this goal. For example, with the LISP Tutor (Anderson and Reiser, 1985), studies showed that while the computer tutor was more effective than “learning on your own”, it was not as effective as a human tutor.

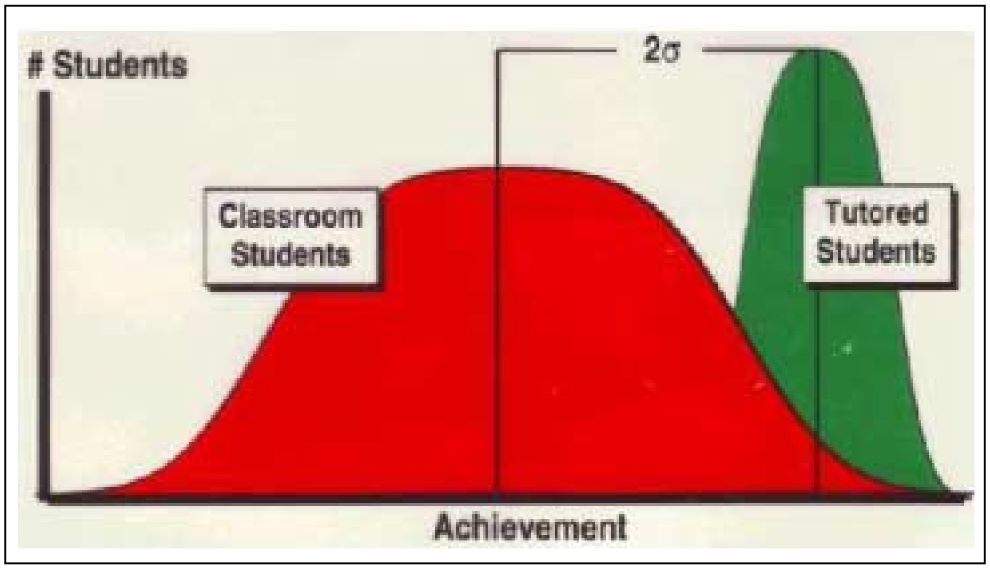

Figure 3. The 2 Sigma Problem (Bloom, 1984)

As mentioned in the previous chapter, there are three types of knowledge used to achieve effective human tutoring – knowledge of the subject matter, knowledge of teaching strategy and methods, and knowledge of the student (Self, 1974). In the following subsection, it will be shown that these three types of knowledge form the building blocks of an ITS.

2.2.1 Components of an Intelligent Tutoring System

There are three important functions of an ITS (McCalla and Greer, 1994). First, an ITS *models* the knowledge of the learner in some computationally useful and inspectable way. Next, based on the student model of the learner, the ITS *intervenes* in the interaction between system and learner with the goal of facilitating learning. Finally, the ITS *evaluates* the success of its intervention and adjusts its model of the learner, and the loop repeats.

In order to carry out these functions in a one-on-one interaction with a learner, an ITS must have a model of instructional content that specifies *what* to teach, a model of a teaching strategy that specifies *how* to teach, and a model of a student that specifies *who* to teach (Ohlsson, 1987). These models are similar to the types of knowledge types mentioned earlier which are necessary for effective human tutoring. Wenger (1987) describes an ITS as a *knowledge communication system* which comprises at least four functional interacting components the domain knowledge model, the pedagogical module, the student model, and the interface or communication module (Figure 4):

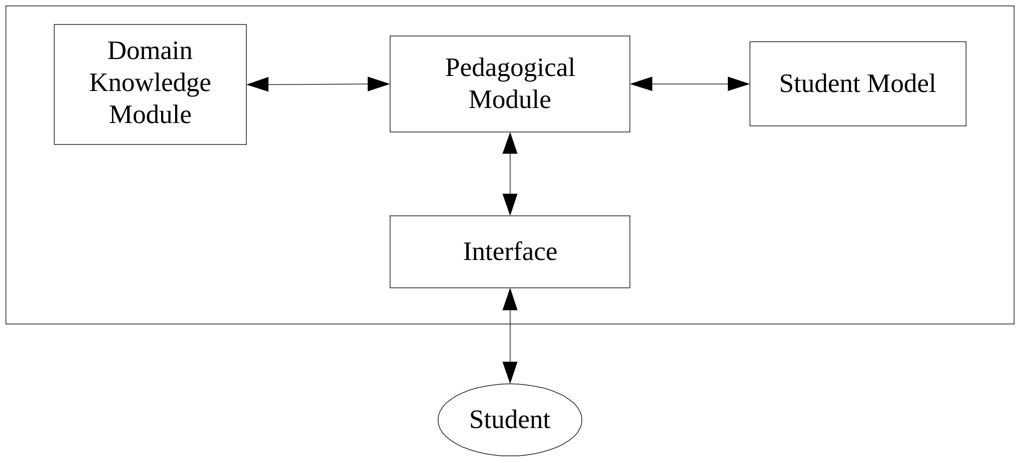

Figure 4. Major Components of an Intelligent Tutoring System

- **Domain Knowledge Module**

This module contains the knowledge of *what* to teach. It represents an area of syllabus and usually requires knowledge engineering in its construction. Domain knowledge is usually represented as skills, concepts, procedures and problems of the subject domain under study.

- **The Pedagogical Module**

This component contains the knowledge of *how* to teach, that is a teaching or tutoring strategy. It orchestrates the whole tutoring process and deals with issues like which topic to present, when to present a new topic, when to present a problem, when to review, and when to offer remedial help.

- **Student Model**

This component contains the knowledge about *who* it is teaching. It keeps track of information that is specific to each individual student, such as his mastery or competence of the material being taught, and his misconceptions. In effect, it stores the computer tutor's beliefs about the student. These information is used by the pedagogical module to tailor its teaching to the individual needs of the student.

- **Interface Module**

This module provides a communication mechanism for handling the interactions between the computer tutor and the student, such as mixed-initiative dialogues.

2.3 Student Modelling and Intelligent Tutoring

Student modelling is a type of *user modelling* and is specifically relevant to the adaptability of intelligent tutoring systems where the users are students or learners. One of the earliest attempts at a student model was by Carbonell (1970) who used a semantic network to represent domain knowledge. The term 'student modelling' stems from CAI research. Early CAI efforts have attempted to enhance individualised instruction through the use of student models. In a seminal paper on student modelling, Self (1974) describes generative CAI systems where problems and comments are not prestored but generated dynamically and this generation is a function of the student model. He further classifies generative CAI systems into two categories where knowledge of what is being taught is kept *implicit* or *explicit*.

While most generative CAI systems belong to the first category, it is only the second category which allows the type of knowledge of who is being taught to refer directly to the type of knowledge of what is being taught, thus allowing the construction of a *student model*. Over the years, the term ‘student modelling’ came to be closely associated with ITSs when increasing efforts by Artificial Intelligence techniques were channelled to tackle the task of student modelling.

Student models can be classified according to the *functions* they can perform. Self (1988b) describes six main functions of student models:

- **Corrective**
  Feedback intended at repairing a misunderstanding of the student. In this case, the model must identify a difference between the student’s understanding and the correct knowledge, and provide this information to other parts of the system.

- **Elaborative**
  Extending the knowledge of the student. In this case, the model should identify areas where the student can be introduced to new material, or a refinement of his current understanding.

- **Strategic**
  Changing the approach to teaching at a higher level than local tactics. This requires the student model to provide more general information about the student, such as his success rate with the current teaching strategy as opposed to a previous teaching strategy.

- **Diagnostic**
  Analysis of the state of the student. In some sense, all aspects of student modelling are diagnostic. What is meant here is the explicit use of the student model to refine information about the student in order to make a decision. If, for example, the tutor wishes to introduce a new topic, but the student model is unable to indicate whether the current level of understanding of the student is adequate, the model can be requested to generate diagnostic examples which can be presented to the student.

- **Predictive**

  Using the model to anticipate the effect of an action upon the student. This requires the student model to act as a 'simulator' to simulate the behaviour of the student, given a particular action.

- **Evaluative**

  Providing an assessment of the level of achievement of the student. This requires the system to make some aggregation across the information that it has.

Early CAI systems, with their branching mechanisms, in some sense supported *corrective* and *elaborative* functions but lacked *strategic, diagnostic* or *predictive* roles, while Carbonell's student model fulfilled all the functions given above.

Student models can also be classified by their *modes* of interpretation: *process* or *state* models (Clancey, 1986). Process models are capable of simulating the process by which the learner solves a problem and can therefore perform the *predictive* function of student modelling. They are also called *executable* or *runnable* models. A student model is executable if its present state can be utilised by a certain interpreter to simulate the behaviour of the modelled student when he is solving a problem. Executable models are also referred to as *procedural* models (Brusilovskiy, 1994) as student knowledge in the student model are usually represented as procedures, the most common knowledge elements being production rules. Examples of process models are LISP Tutor (Anderson and Reiser, 1985) and GUIDON (Clancey, 1979). On the other hand, state models do not have the capability of simulating and contain only state information. Examples include SOPHIE (Brown *et al.*, 1975) for troubleshooting electronic circuits, PROUST (Soloway and Johnson, 1984) for program plan recognition of Pascal programs, and constraint-based models (Ohlsson, 1994).

2.4 Modelling Domain-Specific Knowledge

There are two major types of information which may be contained in a student model: model of domain-specific knowledge and model of individual, learner-related characteristics.

Learner-related characteristics can be *conative* such as wants, intentions and/or *affective* such as motivation, emotion and anxiety (Self, 1994). Examples include modelling motivation (Matsubara and Nagamachi, 1996) and modelling learning styles (Bull and Shurville, 1999). There has been relatively little attention devoted to modelling the learner's state of such characteristics beyond domain knowledge and common misconceptions. As such, there is still a lack of standardised means of classifying these models. On the other hand, there are established techniques for modelling student knowledge in relation to domain or course knowledge. The following subsections discuss these techniques.

2.4.1 Scalar Model

The simplest form of a student model is a *scalar* model, which estimates the level of user knowledge of the course material by means of a certain integral estimate such a number ranging from 1 to 5.

2.4.2 Overlay Model

The *overlay* model is a student model which contains the student's knowledge as a *subset* of the expert or domain knowledge (Figure 5). It works on the basis that students will learn the domain and gain knowledge through aspiring to become experts. Knowledge is represented and structured in the same way for both the domain knowledge and the student model, the difference being in terms of completeness. Knowledge representation techniques include rule-based representations and semantic networks. During student modelling, diagnosis takes place by comparing the student's knowledge with the domain knowledge and the difference is explained as the student's lack of skill.

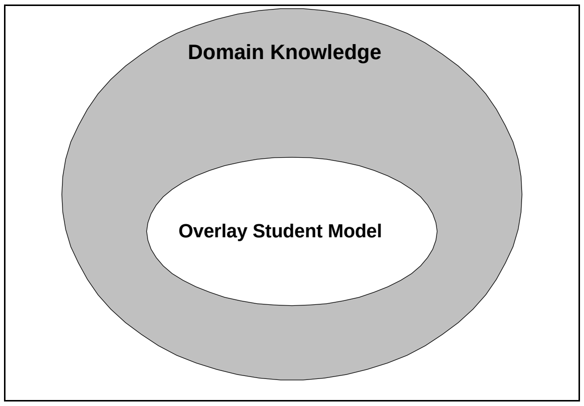

Figure 5. Overlay Student Model

Examples of overlay models are SCHOLAR (Carbonell, 1970), a geography tutor for South America, which adopts a semantic network representation, and GUIDON (Clancey, 1979), a tutor built on MYCIN, a knowledge-based program that provides consultations about infectious disease diagnosis and therapy.

This method is *incomplete* because only the lack of knowledge can be modelled. The main problem with the overlay model is that it assumes that a student's knowledge can be merely a subset of that of an expert, which may not be the case. The domain model is usually represented in terms of atomic units, that is, a student either knows or does not know a certain unit. A student's partial knowledge of a unit cannot be represented. Also, it does not represent any knowledge or beliefs, such as misconceptions, that the student might have that differ from those of the expert. There is no possibility of allowing the student (novice) to have different conceptions of the domain from that of the expert, although there are studies which have shown otherwise. For example, when categorising problems, novices tend to rely on surface analogies between problems while experts use deeper functional analogies (Chi *et al.*, 1981).

2.4.3 Differential Model

The *differential* model is seen as an improvement to the overlay model. It does not assume that gaps in student knowledge are all undesirable. It divides the student's knowledge into two categories: knowledge that the student should know and knowledge the student could not be expected to know (see Figure 6).

Examples of systems which use this approach to student modelling are WEST (Burton and Brown, 1985), an electronic board game to teach arithmetic, and GUIDON2 (Clancey, 1987). This model still suffers from most of the same difficulties as the standard overlay model as it still assumes that the student model is essentially a subset of the expert and the student model remains incomplete.

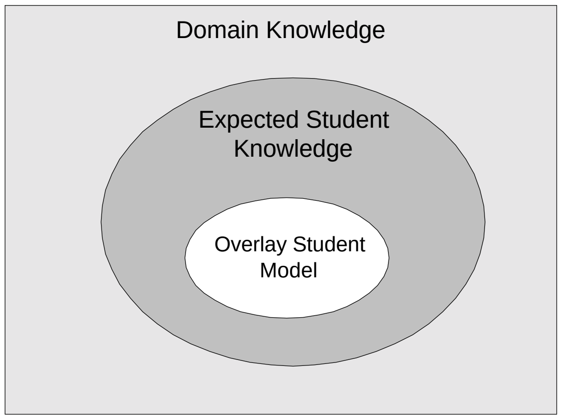

Figure 6. A Differential Student Model

2.4.4 Perturbation Model

The *perturbation* model approach, also called the *buggy* model and the *mal rule* model, goes beyond inferring what the student knows and does not know about a domain by inferring as well, any faulty knowledge or misconceptions that the student might possess. The perturbation student model, which represents the student's correct and faulty knowledge, is considered a subset of both the domain knowledge and buggy knowledge (see Figure 7).

This approach combines the standard overlay model with a representation of faulty or buggy knowledge. The domain or expert knowledge is first represented and then augmented with explicit knowledge of possible misconceptions of the student. This explicit knowledge is known as buggy knowledge and allows a more sophisticated diagnosis of the student's state of knowledge than can be accomplished with a simple overlay model. Subsequent remediation goes beyond filling in gaps in the student's knowledge where the tutor must identify and eliminate the student's misconceptions as well as adding the correct conceptions to the understanding of the student.

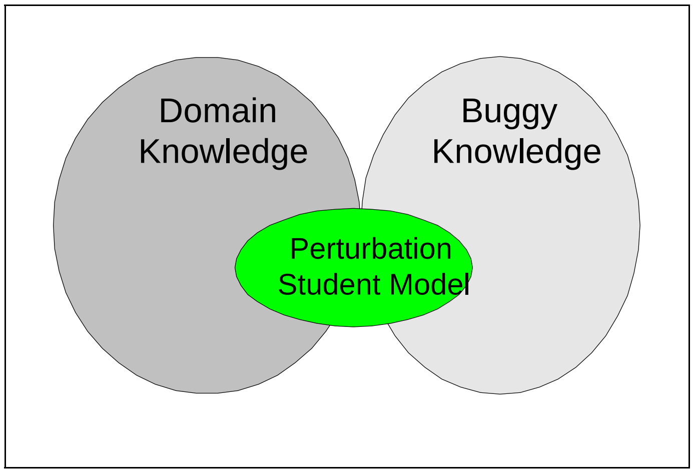

Figure 7. A Perturbation Model

Examples of this approach are DEBUGGY (Brown and Burton, 1978) , a system to evaluate a learner's subtraction performance, Leeds Modelling Systems or LMS (Sleeman and Smith, 1981), a system for testing algebra skills, and PROUST (Soloway and Johnson, 1984), a system for PASCAL programming, and the Geometry Tutor (Anderson *et al.*, 1985).

In the literature on perturbation models, different terms have been used such as *bug*, *misconception*, *error* and *mal rule*. These terms are often used interchangeably but are actually quite distinct from one another. When a student demonstrates a consistent but incorrect general model, this is called a *misconception*. A *bug*, on the other hand, refers to some structural flaw in a procedure that often manifests itself in faulty behaviour. A collection of bugs and misconceptions is referred to as *buggy knowledge, a bug library* or *a bug catalogue.*

When the idea of perturbations is applied to rule-based representations, the buggy knowledge becomes known as *buggy rules* or *mal rules.* An example is BUGGY (Brown and Burton, 1978) where a rule-based domain knowledge of multi-column subtraction was developed together with a rule-based bug library. The bug library was based on a large empirical study where a set of students' responses to a mathematics test were collected and analysed. The correct responses formed the rules in the domain knowledge while the incorrect responses were represented as mal rules in the bug library. In this way, a student's current subtraction procedure could be explained by either a correct rule or mal rule.

The terms *error* and *bug* will now be distinguished. As mentioned earlier, with the perturbation model approach, there is a set of correct procedures which make up the domain knowledge and a set of incorrect or faulty procedures which make up the buggy knowledge. A procedure, when executed, produces a specific behaviour or outcome. If the outcome does not correspond to that what is expected, the difference between expected behaviour and observed behaviour is described as an *error*. A *bug*, however, is a variant of the procedure which generates an error. It is possible for the same error to be explained by different bugs.

Another example of a rule-based representation of correct and buggy knowledge is the LISP Tutor (Anderson and Reiser, 1985) which is based on the *model tracing* technique (Anderson *et al.*, 1990). Under the model tracing paradigm, the computer tutor monitors the student in a step-by-step fashion during problem solving and inferences from single problem solving steps to single rules. In this way, the tutor can determine if the student is on a correct solution path or an incorrect one. In the case of the latter, an attempt is made to match the incorrect solution path against a bug library in order to infer if the student is suffering from a known misconception. Appropriate feedback is provided accordingly. While model tracing prevents floundering on the part of the student, it discourages exploratory behaviour.

There are many challenges to the buggy technique. For example, there is the problem of *bug migration* (VanLehn, 1982) which is caused by the change of a bug into a different but related one and this makes the diagnosis of student’s actions even more difficult. Also, the construction of bug libraries often involves extensive empirical studies including protocol analysis. The high costs involved could be offset by its portability across student populations in a similar subject domain. However, Payne and Squibb (1990) questioned the generality of bug libraries when they conducted a study which showed that the bug library constructed by Sleeman (1984) and Sleeman (1985) were minimally relevant for the two new student populations. Some researchers have attempted to avoid collecting bugs through empirical observations by automating the generation of buggy knowledge (Baffes and Mooney, 1996), (Lee, 1996); another good example is the Repair Theory (Brown and VanLehn, 1980) which is a generative theory of bugs, that is, a method of deriving bug libraries directly from correct procedures. The usefulness of maintaining bug libraries was also questioned by Sleeman *et* *al* (1989) who conducted a study in the tutoring of algebra and found that *reteaching* was as effective as remediating errors.

2.4.5 Genetic Graph

While the first four models described above capture a *snapshot* of the current knowledge of the student, the *genetic graph* (Goldstein, 1982) captures an *evolutionary* process of the student's knowledge over time. It is a type of semantic network which represents the expert's conception of the domain.

The *nodes* of the graph represent the student's knowledge and the *edges* represent the expert's view of how learning occurs between nodes. A student's learning behaviour is shown by a particular learning path in a sequence which corresponds to the genetic graph's partial ordering and this learning path forms part of the student model.

Here, student modelling is still an overlay technique where the student model is a subset of the domain knowledge. The main difference between the standard overlay technique and the genetic graph is that the latter is not only concerned with maintaining what the student knows but also how his knowledge is acquired over time. This is represented by the student's learning path which evolves as he progresses in his learning.

2.4.6 Bounded Model

A *bounded* model (Elsom-Cook, 1988) can be considered a variation of an overlay model. This technique moves away from representing knowledge to working in terms of beliefs about the student's knowledge. The idea is that, rather than attempting to build an exact student model, the student's knowledge is represented by fuzzy bounds. By observing student behaviour, the system maintains a confidence interval around the lower and upper bounds of the student's knowledge. Standard machine learning techniques are used. The lower and upper bounds are obtained through inductive reasoning. Then, on the basis of the system's domain knowledge, deductive reasoning is used to generate predictions and problems are generated to test these predictions. Bounded models can be more tractable to build than exact models but subsequent remediation is less precise.

2.4.7 Constraint-based Model

A *constraint-based* model (Ohlsson, 1994) features as being computationally simple. It does not require large empirical studies for constructing a bug library, nor a runnable expert model or an ideal student model. No computationally expensive inference algorithm is required – simple pattern matching is used. The domain knowledge is elicited through task analysis and is represented as a set of constraints that capture the central concepts of the domain. The student model is the set of constraints which he violates. These violated constraints become candidates for concepts which the student does not know and is used to guide remediation or feedback. An example of this approach is the SQL Tutor (Mitrovic, 1998) which elicited from an expert around five hundred constraints.

Constraint-based modelling does not prescribe a particular tutorial strategy. It ignores the student's problem solving strategy and is thus able to monitor free exploration and to recognise creative and novel solutions as correct. Ohlsson (1994) coined the term *radical strategy variability* which claims that a student has several strategies at each moment in time, and he may switch between them on a problem-by-problem basis. In the face of such inconsistencies or contradictory behaviour observed in student solutions, constraint-based modelling approach fared the best in tackling this problem while the bug library and machine learning techniques fared the worst.

2.4.8 Machine Learning

Machine learning is a technique of Artificial Intelligence which develops computational theories of learning processes and builds machines which learn. Gilmore and Self (1988) examined the potential of machine learning for building student models. A bottom-up approach is adopted which first identifies a solution path that leads to the final answer and then machine learning is applied to perform *reconstructive diagnosis* in order to construct a procedure that generates that path. While this technique does not require empirical research to construct a bug library, the computational complexity involved can even be higher than that of the bug library technique.

Examples include the Tutor of Logically Aided Construction or TALC Geometric Tutor (Desmoulins and van Labeke, 1996) which uses machine learning techniques to check the correctness of student construction with respect to a teacher's specification and the Automated Cognitive Modeler or ACM (Langley and Ohlsson, 1984) which tries to model behaviour & uses machine learning to construct a student model off-line. ACM employs a bottom-up approach which invents a student model by searching the space of possible models, identifying a solution path that explains the performance and constructing a procedure that generates that path; the second step is done by ID3 machine learning method.

2.4.9 Computerised Adaptive Testing

Computerised adaptive testing or CAT (Wainer, 1990) is a recent arrival into the scene of student modelling. With its roots in psychometric measurement, CAT is characterised by the efficiency and accuracy at inferring a student's knowledge in a domain with the minimum number of problems. The student is presented with problems of appropriate difficulty. This has the advantage of reducing test anxiety, sustaining the motivation of students during testing, and more importantly, of reducing the overall testing time. An interesting analogy between measurement within a tutoring system and psychometric measurement was made by Frederiksen and White (1990).

Two major techniques of computerised adaptive testing have been particularly influential in their application in student modelling. They are the Item Response Theory or IRT (Wainer and Mislevy, 1990) and the Knowledge Space Theory or KST (Falmagne *et al.*, 1990). These are discussed in more detail in Sections 2.5.1 and 2.5.2. CAT can be viewed as an overlay technique. In the approach based on the two major techniques mentioned above, the domain knowledge is represented as a problem domain which contains problems or classes of problems for a particular area of syllabus. For example, in the KST approach, the domain may be represented by a directed graph of nodes where each node represents a problem or a class of problems and the edges represent the relationship between the nodes. The student model is a subset of the graph and represents the student's knowledge as a particular path on the graph. Other works have represented the domain knowledge to include not only problems or classes or problems but also concepts and skills. Examples include granularity hierarchies (Collins *et al.*, 1996) and curriculum hierarchies (Huang, 1996).

The following section is devoted to discussing adaptive testing in detail.

2.5 Computerised Adaptive Testing

Traditional fixed-length, pencil-and-paper fixed-item testing, or FIT for short, remains the predominant testing strategy in educational and training settings. FIT involves the administration of a fixed set of questions to a student population. An examinee[1] is expected to answer all questions within a fixed period of time. There is a predefined ordering of questions but an examinee does not need to answer in that order; he may skip questions and return to them later. As this type of testing has to reach out to examinees of all capabilities within a population, therefore, there may be relatively few questions which are of the appropriate difficulty for any one examinee. Questions may be too difficult for the weak examinee, or too easy for a good examinee. As a consequence, large numbers of questions may be needed to obtain an acceptable degree of precision. Also, questions are arranged in order of difficulty. This may work well for a less proficient examinee as he will be able to answer the earlier questions which are easier before reaching the more difficult ones. For a good examinee, however, he would have to wade through the easier ones before reaching the more challenging questions. In both cases, there is a possibility of extraneous noise such as guesswork and careless slips. For the less proficient student, anxiety may set in when he attempts to tackle the more difficult questions and he may attempt to solve them through guessing. For a more proficient student, boredom may set in when he wades through the easy questions and this wastes time and may increase the possibility of noise mainly caused by careless errors or slips.

For the same purposes, a useful variation would be for all examinees to take tests that are individually suited to their own abilities. With the advent of more powerful and affordable desktop computers in the 1980s, it became possible to implement *computerised adaptive testing*, or CAT for short, in educational and training settings. The strength of CAT lies in having to ask only enough questions in order to assess a student in a subject domain and in its ability to rank all examinees on the same continuum even when the examinees do not share any test items in common. CAT is defined as a sequential form of individual testing in which successive items in the test are selected based on an algorithm which adapts the test to the specific characteristics of each examinee. As mentioned earlier, the goal of CAT is to use the

[1] The words ‘student’ and ‘examinee’ are used interchangeably.

least number of questions necessary to determine, with high accuracy, the level of performance of the examinee (Welch and Frick, 1993). An examinee is given an easier problem to solve when he has answered the current one incorrectly, or a more difficult one when he has answered the current one correctly. In this way, an examinee is presented with problems of appropriate difficulty throughout the test. This careful tailoring and selection of problems not only result in greater accuracy of the assessment with only a handful of properly selected items but also reduces the overall testing time.

An adaptive test is usually computerised, although a manual method may be used such as the *self-scoring flexilevel test* described by Frederic Lord in 1971, a description of which is given by Thissen and Mislevy (1990). The main characteristics of CAT are:

- The test can be taken at the time convenient to the examinee; there is no need for mass or group-administered testing, thus saving on physical space.
- As each test is tailored to an examinee, no two tests need be identical for any two examinees and this minimises the possibility of copying.
- Questions are presented on a computer screen one at a time.
- Once an examinee keys in and confirms his answer, he is not able to change it.
- The examinee is not allowed to skip questions nor is he allowed to return to a question which he has confirmed his answer to previously.
- The examinee must answer the current question in order to proceed onto the next one.
- The selection of each question and the decision to stop the test are dynamically controlled by the answers of the examinee.

Examples of the use of CAT include three of the world's most widely used college and graduate admissions tests which are trading the pencil-and-paper formats for CAT (Educational Testing Service, 1999;Oseas-Europe, 2000). These are Graduate Record Examinations (GRE®) General Test (GRE, 2000), the Graduate Management Admission Test GMAT® (GMAT, 2000), and the Test of English as a Foreign Language TOEFL® (TOEFL, 2000). Other major moves include Microsoft® for the Microsoft® Certified Solution Developer (MCSD) credential (Microsoft, 2000), and COMPASS®/ESL which measures students' mathematics, reading, and writing skills on demand (COMPASS, 2000). Smaller scale examples include a commercial application called SWIFT in the domain of desktop computer applications (Gemini, 2000) and a computer tutor called Mathemagic in the domain of mathematics (Parvate *et al.*, 1998).

A variation of CAT is self-adaptive testing, or SAT for short, where an examinee can exercise some control over the sequencing of problems. The examinee, rather than a computerised algorithm, chooses the difficulty of the next problem to be presented (Rocklin, 1994). In a study carried out by Rocklin and O'Donnell (1987), SAT was compared against the more traditional FIT. Participants completed a self-report of text anxiety and were randomly assigned to take one of the three tests of verbal ability. *Anxiety* is associated with decrements in academic performance and is typified by a situation where a student claims to have mastered the course material before the test or examination but is unable to perform satisfactorily during the test, only to recall the material with complete clarity after it is too late (Covington and Omelich, 1987). The study showed that SAT not only led to higher ability estimates but also minimised the effect of test anxiety without any overall loss of measurement precision.

A more thorough study which compared all three testing procedures – FIT, CAT and SAT – was conducted by Vispoel *et al*. (1994). In this study, three aspects of the tests were evaluated: (a) their measurement precision and efficiency, (b) the effects of several individual difference variables (test anxiety, verbal self-concept, computer usage, and computer anxiety) on ability estimates alone and in interaction with the test administration procedures, and (c) examinees' attitudes about various features of the tests they took. Volunteer college students were assigned to take a vocabulary test using one of the three methods of testing. All the tests used the same large, well-calibrated item bank. The results were interesting. CAT was found to be more precise and efficient than SAT, which was in turn more precise and efficient than FIT. SAT, however, yielded higher ability estimates than the other tests for individuals with lower verbal self-concepts. Examinees indicated that they prefer tests in which they can have as much control and as much information as possible. Taken collectively, these results indicate that no single test administration procedure simultaneously maximises precision, efficiency, validity, and examinee satisfaction. As each testing procedure has its own benefits, some systems make use of more than one testing procedure. For example, the work of Frosini *et al.* (1998) combined both CAT and SAT in their creation of an automatic examiner in the domain of Pascal programming, where a SAT pre-exam determined the starting difficulty level of the CAT.

Despite its many advantages over FIT, CAT is not as widely used in the educational or training environments as would be expected. A reason for it not to be readily embraced may be that it is still a relatively new approach when compared with the traditional FIT which have been the predominant testing strategy for generations. Also, mass testing, a feature of FIT, is still favoured because it is a relatively cheaper form of test administration.

Unlike ITSs, most CAT systems describe a subject syllabus in terms of test items or problems only, the main sources are likely to be from teachers, instructional materials and past class or exam questions. A *problem*, as defined in the Oxford English Dictionary, is "a doubtful or difficult question; a matter of inquiry, discussion, or thought; a question that exercises the mind". Problems must be chosen very carefully (Marshall, 1990) as they determine the efficiency and effectiveness in carrying out assessment to determine what a learner knows, understands, and can do in order to further learning and performance. Also, CAT makes more stringent demands on its component items than its FIT counterpart. This is because its tests are relatively shorter, usually half as long as FIT but with the same degree of accuracy, and therefore each item is critical and must be well constructed. The effect of a *flawed* item in CAT has much greater impact than one in FIT, mainly because not every examinee gets the same test so a flawed item may affect some examinees but not others – this compromises on test fairness. In the next two sections, the two major approaches of CAT are described, based on two aspects – the building of the domain of test items and the problem progression strategy.

2.5.1 Item Response Theory

Item Response Theory (Wainer and Mislevy, 1990), or IRT, is a statistical framework in which examinees can be described by a set of ability scores that are predictive, linking actual performance on test items, item statistics and examinee abilities. IRT was first proposed by Lord (1980) and is well explained by Wainer (1990). There are web-based tutorials on IRT (Rudner, 1998). True to the goal of CAT in general, IRT-based adaptive testing systems have been shown to significantly reduce testing time without sacrificing reliability of measurement (Weiss and Kingsbury, 1984).

Ideas from IRT have been very influential in student modelling and intelligent tutoring. They have formed the basis of a system to assess student programming abilities (Syang and Dale, 1993), they have influenced Huang's content-balanced tests (Huang, 1996), and more recently, their influence can be seen in SIETTE which is a web-based adaptive testing system in the domain of European vegetable species (Ríos *et al.*, 1999).

2.5.1.1 Describing the Domain

In the terminology of IRT, a domain of test problems or items is called an *item pool*. Each item is associated with one or more of these parameters – the *difficulty level*, the *discriminatory power* and the *guessing factor*. The *difficulty level* describes how difficult an item is, the *discriminatory power* describes how well the test item discriminates students of different proficiency while the *guessing factor* is the probability that a student can answer the item correctly by guessing.

To ensure that it best fit the current student population to be tested on, an item pool must undergo content-balancing and item calibration. Content-balancing is used to ensure no content area is over-tested or under-tested. Item calibration is used to estimate values for the item parameters. This process is expensive as it involves large-scale empirical studies, usually based on a minimum of 200 to 1000 or more students. An effort to avoid major empirical studies for item calibration is the work by Huang (1996) whose CBAT-2 algorithm uses a machine learning procedure to generate content-balanced questions based on a specific part of a course curriculum.

2.5.1.2 The Problem Progression Strategy

The problem progression or adaptive testing algorithm in IRT (Thissen and Mislevy, 1990) is given in Figure 8. At the start of the test, the algorithm has an initial provisional proficiency estimate of the student and this is denoted by θ. This specifies an initial item which is selected from the item pool. The selected item is aimed at providing the most information about the student; the notion of the "most informative" item will be discussed later. Once the student provides an answer for the selected item, a new proficiency estimate, θ', is calculated together with its confidence level. It is based on whether the student's answer is correct or incorrect, the old θ and the item parameters. If the confidence level of θ' reaches a designated level, or when some predetermined number of items has been administered, the test terminates. Otherwise another item is selected for the student, and the test continues.

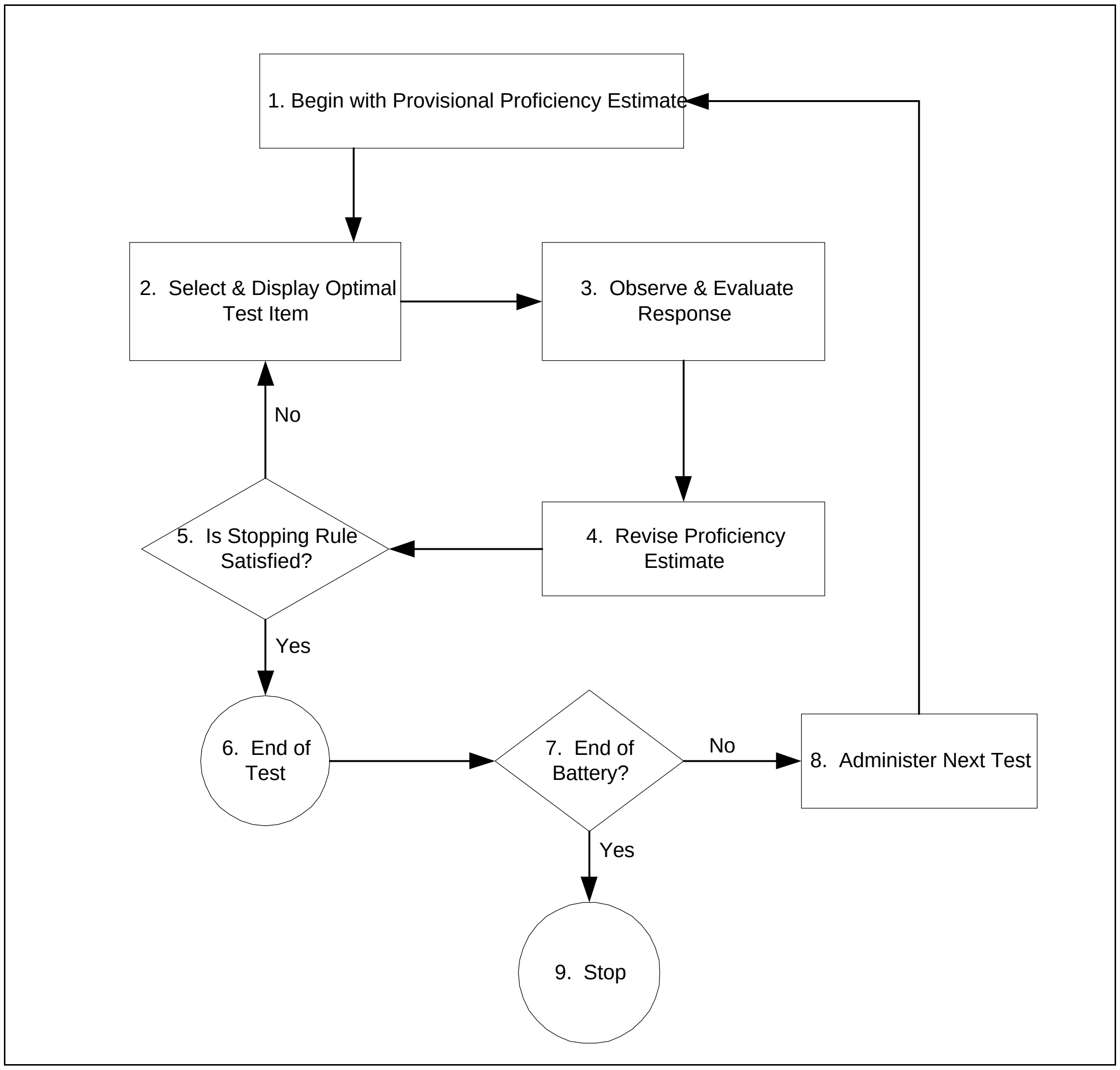
1. Begin with Provisional Proficiency Estimate
2. Select & Display Optimal Test Item
3. Observe & Evaluate Response
4. Revise Proficiency Estimate
5. Is Stopping Rule Satisfied?
No
Yes
6. End of Test
7. End of Battery?
No
Yes
8. Administer Next Test
9. Stop

Figure 8. A Flowchart describing an Adaptive Test (Thissen and Mislevy, 1990)

As mentioned earlier, an item does not need to be characterised by all three parameters. For example, problems in CBAT-2 (Huang, 1996) are indexed by two parameters – difficulty level and guessing factor. Therefore, depending on the number of parameters used, the confidence level or probability of θ' is calculated by any one of the three formulas where $\boldsymbol{b}$ is the difficulty level, $\boldsymbol{a}$ the discriminatory power and $\boldsymbol{c}$ is the guessing factor (Wainer and Mislevy, 1990). These formulas are presented in Figure 9.

$$P(\theta) = \frac{1}{1 + e^{-(\theta - b)}}$$ for one parameter. Also called one parameter logistic model or 1-PL or Rasch model

$$P(\theta) = \frac{1}{1 + e^{-a(\theta - b)}}$$ for two parameters. Also called 2-PL model

$$P(\theta) = c + \frac{1 - c}{1 + e^{-a(\theta - b)}}$$ for three parameters. Also called 3-PL model

Figure 9. Formulas for 1-PL, 2-PL and 3-PL models

P(*θ*) is the probability of the examinee with proficiency *θ* responding correctly to an item. As mentioned earlier, an item that is chosen is one which provides the most information about the student. Items which offer the most information are those whose P(*θ*) equals or are close to 0.5. The reasoning behind this is as follows: If P(*θ*) is more than 0.5, say 0.85, then the test item would not be very informative because it is almost certain that the student would provide a correct response to that test item. If P(*θ*) is less than 0.5, say 0.1, then the test item is also not very informative as it can be fairly certain that the student would respond incorrectly. If P(*θ*) is 0.5, then the test item is considered the most informative item as there would be an equal chance of the student answering correctly or incorrectly.

Item characteristic curves or ICCs can represent the 1-PL, 2-PL and 3-PL models for different values of a, b and c, as shown in Figures 10, 11 and 12 respectively. Appendix A presents more ICCs for the 2-PL and 3-PL models for values of ***b*** equal to 0 and -1. A detailed account of each model is given by Wainer and Mislevy (1990). Figure 10 shows three ICCs representing three different values of difficulty, ***b***. Figure 11 represents three ICCs for three different values of the discriminatory power, ***a***, with the value of ***b*** constant at 1. Figure 12 represents three ICCS for three different values of ***a***, with the values of b and c (the guessing factor) constant.

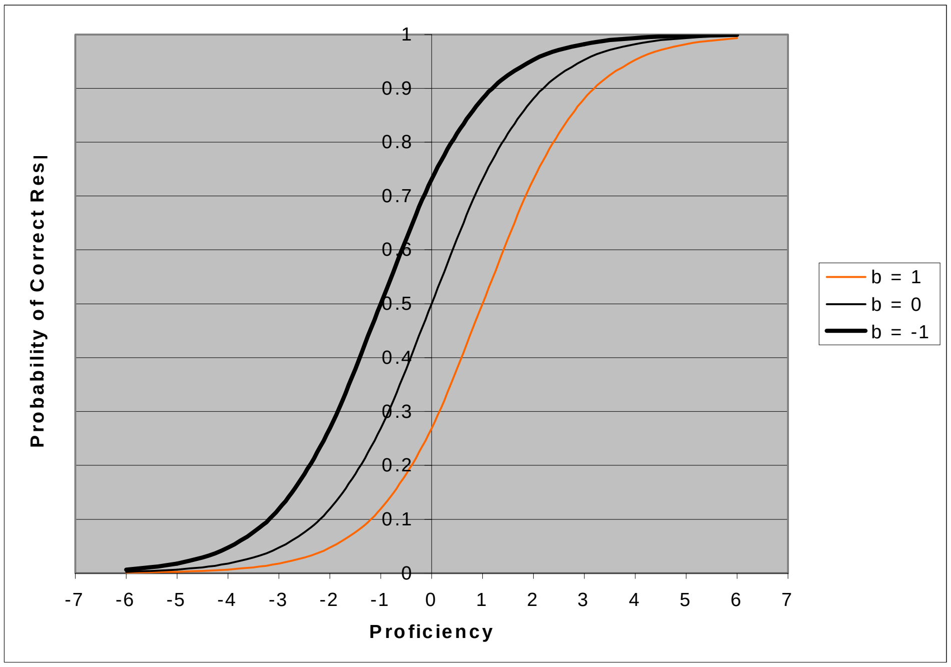

Figure 10. Item Characteristic Curves for 1-PL Model at three levels of difficulty

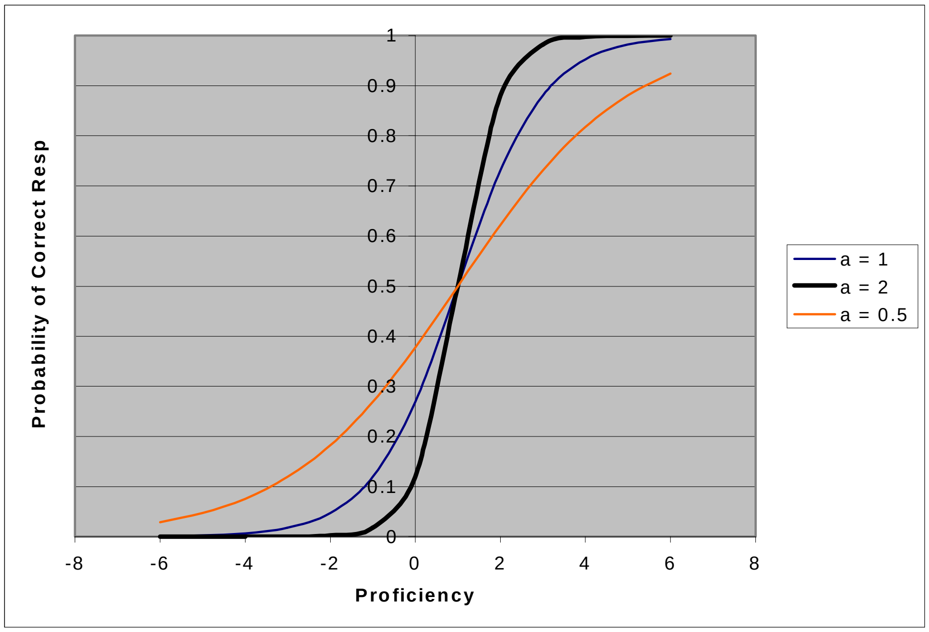

Figure 11. Item Characteristic Curves for 2-PL Model (with difficulty level b=1)

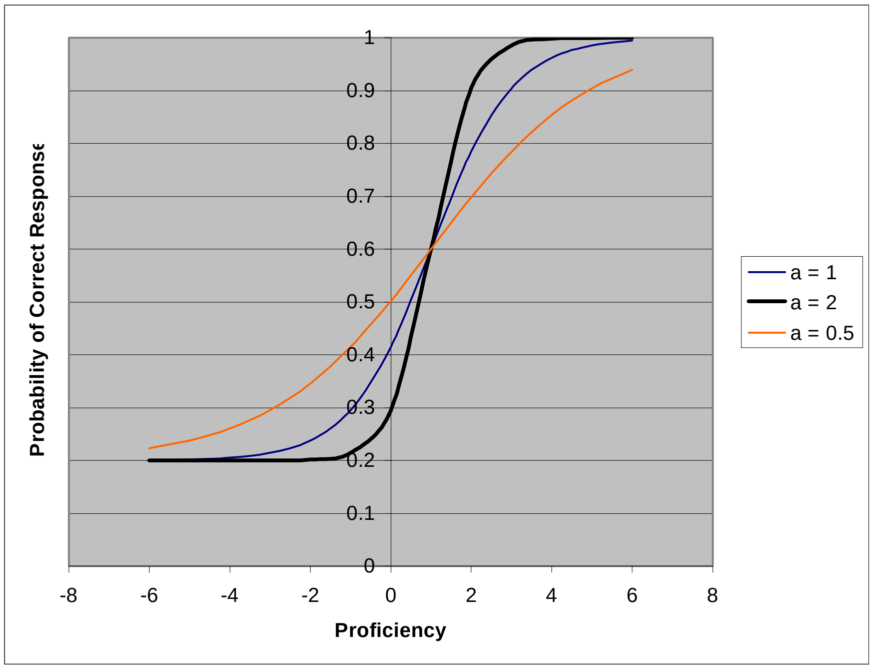

Figure 12. Item Characteristic Curves for 3-PL Model
(with difficulty level b = 1 and guessing factor c = 0.2)

Each item in the item pool is represented by an item characteristic curve or ICC which is determined empirically using an item calibration procedure. As an example, assume that each item in an item pool is represented by any one of three curves in Figure 10. If the student's proficiency estimate θ is 0, then the probability of the student getting an item of difficulty level 0 (b = 0) correct is 0.5, the probability of him getting an item of b = 1 (a more difficult problem) correct is about 0.28, and the probability of him getting an item of b = -1 (an easier problem) correct is about 0.73. As mentioned earlier, an item whose probability estimate is close to or equal to 0.5 is the one which provides the most information about the student and is thus chosen to be presented next to the student.

2.5.2 Knowledge Space Theory

Another strand of development in adaptive testing is based the Knowledge Space Theory, KST for short, (Doignon and Falmagne, 1985), (Falmagne *et al.*, 1990). Examples of applications include a web-based, domain-independent system called RATH (Hockemeyer and Dietrich, 1999), a web-based system for the domain of mathematics called ALEKS (Doignon and Falmagne, 1998) and a general purpose system for testing and training called ADASTRA (Dowling *et al.*, 1996).

2.5.2.1 Describing the Domain

As with IRT-based systems, the domain is defined by a collection of test items. Here, a test item can represent not only a problem but also a class of problems, and the relationship between the test items are explicitly stated through prerequisite relationships. Unlike IRT-based systems which are *unidimensional* in that only one student trait (such as mastery of a topic) can be measured at one time, adaptive testing systems based on the KST can measure more than one trait and can represent a set of skills or problems mastered by the student. This set is known as a *knowledge state*. The structure of the domain takes the form of a *knowledge space* which represents the area of the syllabus to be tested; the following example will explain the notion of a knowledge space.

A body of knowledge is characterised by a set of items called the domain, say {a,b,c,d}. This gives rise to 2^4 possible knowledge states:

{}, {a}, {b}, {c}, {d},
{a,b}, {a,c}, {a,d}, {b,c}, {b,d}, {c,d},
{a,b,c}, {a,b,d}, {a,c,d}, {b,c,d}, {a,b,c,d}

A student's *knowledge state* is defined as the set of items in the domain that the student is capable of solving. For example, if a student has the knowledge state {a,b,d}, this means that he can solve items a, b and d. Not all possible subsets of the domain are *feasible* knowledge states. For example, if the student can solve item d, and that it is inferred that the student can also solve item a, then any knowledge state that contains item d must also contain item a. This means that knowledge states {d}, {b,d}, {c,d} and {b,c,d} are not feasible. This means that a feasible knowledge state is one which contains not only all the items that the student has demonstrated mastery of, but also the items which can be inferred. In effect, a feasible knowledge state describes the *prerequisite relationships* between items. For example, in the knowledge state {a,d}, item a is a prerequisite of item d. The collection of all feasible knowledge states is called the *knowledge structure.* The knowledge structure must also contain the *null state {}* which corresponds to the student who cannot solve any item, and the *domain* which corresponds to the student who can solve or master all items.

When two subsets of items are knowledge states in a knowledge structure, then their union is also a state, that is,

If ***K*** and ***K'*** are states, then ***K*** U ***K'*** is also a state

This means that the collection of states is *closed under union*. When a knowledge structure satisfies this condition, it is known as a *knowledge space*.

In practice, feasible knowledge states are obtained through computer-aided procedures which systematically query human experts to obtain their *personal* knowledge structures (Dowling and Kaluscha, 1995) (Dowling, 1993) (Koppen, 1993) (Kambouri *et al.*, 1994). The result of such elicitation is a knowledge space which is a set of all feasible knowledge states. These query procedures present assertions of the form below to the expert teacher for judgement and ask the teacher to either accept or reject each displayed assertion (Dowling and Kaluscha, 1995):

Imagine a student who does not master the items $p_1, \ldots, p_k$.
Is it then (practically certain) that this student does not master item q?

The problem is that the number of possible assertions increases with the number of test items. For example, if there are 50 test items, then there are approximately 2.8 x 10^{16} possible assertions. Not all these assertions need to be presented to the expert for judgement as judgements on assertions whose acceptance or rejection can be *inferred* logically from previous judgements can be omitted. This was tried by Kambouri, Koppen, Villano and Falmagne in a study in which experts judged assertions on 50 examination questions concerning U.S. high school mathematics (Kambouri *et al.*, 1994). In this study, the experts judged between 1000 and 2500 assertions until all 2.8 x 10^{16} assertions were deduced as inferences.

2.5.2.2 The Problem Progression Strategy

The knowledge space will serve as the core of a knowledge assessment system. Once the domain is represented as a knowledge space, the adaptive testing strategy is then to locate as efficiently and as accurately as possible, a student's knowledge state, which is a point in the knowledge space. Problem progression works like this. An item is selected. Usually, some predictive probabilistic model is used to determine the sequence of items in a test (Villano, 1992). If a student has answered the item correctly (incorrectly), it can be inferred that he can (cannot) answer a prerequisite (parent) item and will thus not be asked to solve the latter. An item is a problem from a pool of similar problems, for example, problems which 'add two fractions of common denominators'. Inferences progressively prune the search space and at the end of the test, a student's knowledge of the subject domain is represented by a knowledge state. In an example given by Dowling and Kaluscha (1995), a knowledge space is represented as an AND/OR graph, as shown in Figure 13. The nodes represent problems and the arcs state the prerequisite relationship between the nodes.

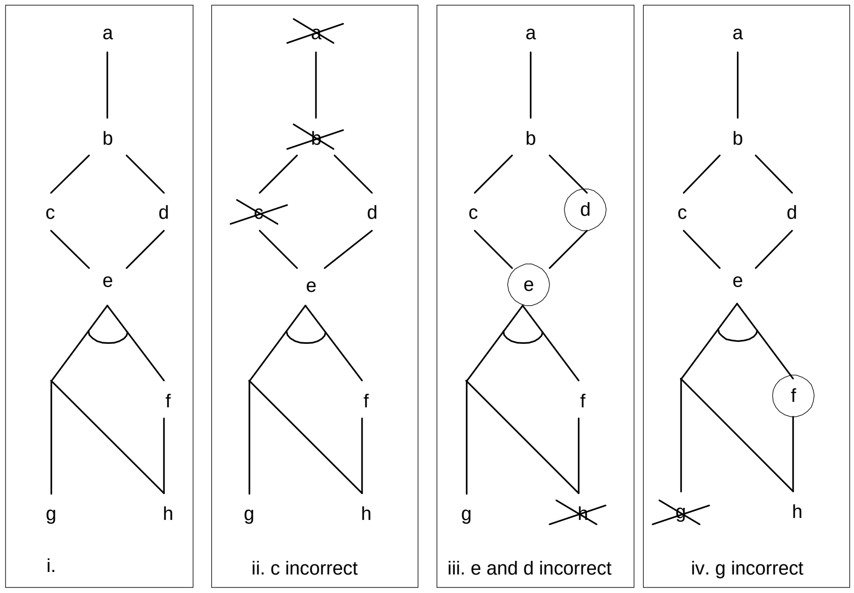

Figure 13. Illustration of Prerequisite Relationships and the Assessment Algorithm

Item *b* represents an AND node. This means that if item b is answered correctly (mastered), then it can be inferred that both its prerequisites *c* and *d* are mastered. Item *e* represents an OR node. This means that if item e is mastered, it infers that all the test items in at least one of its prerequisite subgraphs must be known. These subgraphs are the one with the nodes *g* and *h*, or the one with the nodes *f* and *h*.

Suppose item *c* is chosen and presented to the student and the student provides an incorrect answer. From this incorrect response, it can be inferred that the student will not be able to solve problems *a* and *b* either, as *c* is a prerequisite of *a* and *b.* Problems *a* and *b* are considered more difficult than *c* and are thus not presented to the student. The next problem to be selected will be one which is not *a* or *b*, nor one which has *a, b* or *c* as its prerequisites. Suppose the next problem chosen is *e* and it is answered correctly. This infers a correct answer to problem *h*, as *h* is a member of both subgraphs of the OR node, and *h* will be removed from the list of candidate problems to be presented. Suppose *d* is presented next and is answered correctly, and *g* is presented next and is answered incorrectly. This infers a correct answer to *f.* The test stops and the student's knowledge state is inferred as *{d, e, f, h}* which was reached with only four questions being presented out of a maximum of eight.

2.6 Challenges in Student Modelling

Student modelling remains a difficult task and represents one of the most challenging subfields of ITSs. Some barriers to student modelling which result from the problem of inferring knowledge from learner's behaviour are:

- the environment contains a large amount of uncertainty and noise
- the learner's inference may be unsound and may be based on inconsistent knowledge
- constructing explanations from behaviour is computationally intractable
- learners are creative and inventive and frequently engage in unanticipated, novel behaviour that requires much sophistication to interpret
- There is constant revision which the learner undergoes in his perceptions of the domain of study as the instructional interaction proceeds, a feature that presents a constantly moving target for the student modelling subsystem

As student modelling is an essential component of an ITS, the difficulties associated with student modelling may be a major contributory factor for the relatively slow deployment of

ITSs into educational and training settings. Also, the construction of ITSs, in particular the student modelling task, is a multidisciplinary endeavour and usually involves a team of computer programmers, domain experts and educational theorists. Estimates of construction time indicate that 100 hours of development translates to 1 hour of instruction (Beck *et al.*, 1998). There have been differing views on how the problem of student modelling should be tackled. These are discussed in the following subsections.

a. Shift Away from One-on-One Tutoring

As student modelling seems to be an acute problem in a one-on-one tutoring environment, many researchers have turned to alternative paradigms for student-system interaction in the attempt to avoid the difficulty of doing student modelling. These alternative solutions include collaborative learning, negotiated learning, guided discovery, discovery learning, situated learning and constructivism. However, it is argued that each of these alternative approaches still has a need for student modelling (McCalla, 1992).

b. Abandon the Idea of a Student Model

Due to the difficulty associated with student modelling, even building a partial model is a challenge. Since student models are at best imprecise, it is argued that having no student model at all is better than an inaccurate one. Therefore, some researchers have abandoned the student modelling problem altogether, claiming either that it is intractable or that it is unnecessary or that systems can be effective tutors without such a model (Gugerty, 1997). However, without a student model, an ITS would be doomed to follow a preset sequence of steps regardless of the impact of its actions on a student's learning (Greer and McCalla, 1991). It would be like a human tutor who knows nothing about the individual learner, and therefore is unable to adjust instruction to changes in the learner's behaviour (Holt *et al.,* 1994).

While it is true that the ultimate goal of a completely accurate student modelling system will never be reached and is probably impossible in principle, student modelling is essential to effective intelligent tutoring as the very definition of an ITS as intelligent and individualised is intimately tied to its student modelling capabilities (McCalla, 1992). Many key researchers like Self (1990) and McCalla (1992) have argued that no sensible interactions between a tutoring system and a student can happen without an accurate model or at least generic knowledge of the student's cognitive abilities. McCalla (1992) stressed that student modelling is:

> "not about building exact cognitive models. If it were, we would have to solve all the problems of cognitive science, and teach a machine to be a cognitive scientist, before we could build a student model. We only need to model to the student the level of detail necessary for the teaching decisions we are able to take. If the tutor only has two choices of action, then the model only needs to be accurate enough to distinguish between them."

Self believes that a computer tutor can never have perfect knowledge of a student - some sort of approximation of the beliefs, knowledge, and goals of the student is the best that can be expected. He also believes that this would suffice as there really is no need for a completely accurate student model since even human teachers do not have absolutely accurate perceptions of individual students and yet, they can still teach 'effectively' by employing their partial knowledge of student cognition to good effect, using both generic knowledge of stereotypical student and particular knowledge of individual students whom they are teaching (Self, 1974).

A few recommendations on making student models more tractable were made along the four slogans (Self, 1990):

- *"Avoid guessing*" - design the student-computer interactions such that information needed to build a student model is provided by the student rather than being inferred by the ITS from inadequate data.
- *"Don't diagnose what you can't treat"* - link the proposed contents of the student model with specific instructional actions, ideally supported by educational evidence, in order to clarify what is really needed (and not needed) in the student model.
- *"Empathise with the student's beliefs, don't label them as bugs"* - view the contents of student models as representing the learner's beliefs about the world; the role of the ITS is then to assist the learner in elaborating those beliefs.
- *"Don't feign omniscience - adopt a 'fallible collaborator' role"* - develop ITSs which adopt a more collaborative role, rather than a directive one, for then the style correspond to a better philosophy of how knowledge is acquired (the fidelity of the student model is of less importance).

c. Increase Student Participation

As pointed out earlier, the accuracy of the contents of student models can be increased by enabling the student to explicitly interact with the tutor to actively collaborate on building a student model (Self, 1990). Another solution is to increase student control and participation in order to encourage reflection on the part of the student, by having *inspectable* student models which students can view and argue its accuracy. Examples include the work on stereotypes and scrutable models (Kay, 2000), inspectable learner models (Paiva, 1995) and open learner modelling (Morales *et al.*, 1999).

d. Develop Standards

As research in ITSs and student modelling matures, there is increasing work in developing standardised methods for constructing ITSs, in particular student models. These include the use of authoring tools (Murray, 1999), ontologies (Mizoguchi, 2000;Mizoguchi and Bourdeau, 2000), an actor-based approach (Frasson *et al.*, 1996), a component-based approach (Ritter *et al.*, 1998), plug-in tutor agents (Ritter and Koedinger, 1996), and minimalist design technique (Gutwin *et al.*, 2000).

2.7 Dealing with Uncertainty

In dealing with uncertainty, Bayesian Belief Networks, also known as belief networks and causal probabilistic networks, are fast becoming a popular approach in student modelling.

A good introduction to Bayesian Belief Networks is provided by Jensen (1996), Charniak (1991), and, Russell and Norvig (1995). A Bayesian Network is a directed acyclic graph that organises a body of knowledge in any given area by mapping out causal relationships between nodes and encoding them with prior probability values that represent the extent to which one node is likely to affect another. The nodes represent assertions and an arc from a node A to a node B expresses that A is a cause of B.

Bayesian Belief Networks are based on the Bayes probability theorem (Bayes 1763):

$$P(H \mid E,c) = \frac{P(H|c)*P(E|H,c)}{P(E|c)}$$

where the belief in the hypothesis H can be updated given the additional evidence E and the background context c. The left-hand term, P(H|E,c) is known as the *posterior probability* or the probability of H after considering the effect of E on c. The term P(H|c) is called the *prior probability* of H given c alone. The term P(E|H,c) is called the likelihood and gives the probability of the evidence assuming the hypothesis H and the background information c is true. Finally, the last term P(E|c) is independent of H and can be regarded as a normalizing or scaling factor. A more detailed account is given by Niedermayer (1998).

Bayesian Belief Networks can be used to represent a student model where each node represents a key element of the subject domain. Prior probability values are usually obtained empirically. With new evidence, such as a student's response to a problem, an update algorithm (e.g. Pearl 1988; Neapolitan 1990) is run and each node is assigned a posterior probability value. In the end, when there is no more new evidence, the posterior probability value of each node represents the mastery level of the student.

An extensive survey of Bayesian student modelling is provided by Murray (1998). Key efforts include the works of Collins et al. (1996), VanLehn and Martin (1997) and VanLehn and Niu (2001).

2.8 Conclusion

This chapter reviewed the literature on student modelling in intelligent tutoring systems. In particular, it looked at major student modelling techniques which modelled domain-specific knowledge. Problems associated with student modelling were discussed.

The chapter then discussed computerised adaptive testing in more detail. It first compared computerised adaptive testing with other testing strategies such as fixed-item testing and self-adapted testing. It presented two major approaches of computerised adaptive testing commonly used in student modelling were described: IRT and KST. For each approach, it was shown that the way in which the domain was structured had an influence on problem progression. The review of these two techniques has direct implications for the design and construction of the knowledge components of SKATE.

Firstly, it is the consideration of the basic unit which makes up the domain knowledge or test syllabus. While the basic unit in the IRT and KST approaches is usually a problem or a class of problems, the basic unit of the domain knowledge component in an ITS is usually a description of a body of knowledge in terms of skills, concepts and problems. In SKATE, this will be decided by the expert. The IRT approach maintains large repositories of test items and an attractive option might be to categorise these as classes of problems, as was with the case of KST-based adaptive testing systems. There is a need to generate problem instances from classes of problems, although this was not discussed in the KST approach. These issues are taken up in Chapter 3.

Secondly, the structure of the domain is an important consideration. The development of an IRT test requires a calibration exercise, which can benefit from a large sample size. This is neither feasible nor affordable for SKATE when the aim is to conduct small-scale tests. The KST approach looks like a reasonable candidate for incorporation in SKATE although it has always been discussed by its authors in connection with large (for example, fifty items) tests. This is taken up in Chapter 3.

Thirdly, in problem progression, the key consideration is the selection of problems of appropriate difficulty at all times. IRT employs a statistical model which is used iteratively after a student's response in order to meet the stopping criteria or to select the next item. KST relies heavily on the explicit structure of the domain elicited from one or more experts and problem progression involves the pruning down of a set of candidate problems in the space at each iteration. The issue of problem progression for SKATE is discussed in Chapter 4.

Lastly, the use of a student model is an important issue. In the IRT and KST approaches, there is no explicit mention of the use of a student model to aid item selection and subsequent remediation. In the IRT approach, a single proficiency estimate together with its confidence level is used throughout testing to aid in test item selection and in stopping the test. At the end of the test, this single estimate denotes the student's proficiency in the subject domain. As SKATE is a student modelling framework, more information about the student's mastery of the subject than just a single estimate is necessary. For this reason, the KST approach might be more appropriate as it employs an overlay model where a knowledge state represents a subset of the problem domain and is a list of items mastered by the student.

Chapter 3.

Knowledge Acquisition and Representation

3.1 Introduction

There are several problems to be confronted when adopting an expert emulation approach to designing an adaptive test. Firstly, there is the problem of choosing a suitable expert. Secondly, there is a need for finding suitable knowledge acquisition techniques to aid the elicitation process. Thirdly, there is a need for knowledge representation of the area of syllabus to be tested.

This chapter first lays the foundation for knowledge acquisition by describing the context in which the research is conducted. It includes a description of a small experiment called MATT which confirms that the expert teacher performs adaptive testing when assessing a student's knowledge in a subject domain on a one-on-one interaction. Knowledge acquisition techniques are discussed including the potential use of constraint logic programming in numeric domains.

The chapter then discusses the results of knowledge elicitation in the construction of the domain knowledge which represents the test syllabus. Other issues like identifying problem solving skills, generating problems, diagnosing student answers and measuring problem difficulty are discussed. It also shows how Constraint Logic Programming can be used for knowledge elicitation, knowledge representation, problem generation and answer evaluation. This versatility comes from the dual nature of logic programming. The declarative aspect of clp(FD) facilitates the definition of classes of problems while the procedural aspect allows sample problems to be generated.

3.2 Context

This section describes the context in which expert emulation is carried out. It describes the choice of experts, the type of students under study, the choice of domain, and the role of the expert in his interaction with his students.

3.2.1 Choosing an Expert

The task of finding a suitable expert is not an easy one. Firstly, the expert must be *willing* to participate in the elicitation process (Lightfoot, 1999). Secondly, there is the task of distinguishing a skilled tutor from a novice, in terms of experience in teaching and tutoring. For example, studies have shown that strategies of an expert differ from a novice in tasks such as problem categorisation (Chi *et al.*, 1981) and tutoring (Glass *et al.*, 1999). For example, in a study conducted by Glass, Kim, Evens, Michael and Rovick on the CIRCSIM Tutor, it was found that expert tutors are more likely than novice tutors to query students for information as opposed to informing them directly.

The validity of expert systems depends on the quality of the emulated expert. For the experimental work described in this thesis, an experienced teacher was selected as an expert. He has been in the teaching profession for over ten years and has taught Mathematics and Physics at various UK educational institutions. He is accustomed to interacting with students in a classroom setting as well as on a one-to-one basis. He has taught students from widely differing backgrounds and capabilities, ranging from gifted students to those with learning difficulties. At the time of this study, the expert was teaching remedial mathematics to a population of male adults who are serving time at a local prison.

3.2.2 Type of Students

The type of student population under study is a non-typical one. This is a population of male adult prisoners who do not share many of the characteristics of conventional classroom students, but may tend to have more in common with adult learners engaged in distance learning. The key characteristics of this type of population are:

- the transient nature
- diverse educational and cultural backgrounds
- varying levels of prior knowledge in subject domain
- generally low academic achievements
- low motivation and confidence levels

3.2.3 Choosing a Domain

The mathematics curriculum for this student population is one that prepares the students for UK qualifications in City & Guilds (Key Skills), City & Guild (Number Power), and for GCSE level examinations. In these courses, students must demonstrate the ability to solve mathematics problems given in instructional texts and apply problem solving skills to handle mathematical tasks in every day situations such as calculating tax, writing cheques, and working out bills such as electricity and gas.

Mathematics has many applications in real-life situations and for this reason, it holds an important part in educational curriculum. There is however usually a gap between the ability to solve classroom mathematics problems and the ability to use the same skills to solve real life mathematical tasks. Cooper and Dunne (2000) conducted a study on six hundred students in the age group of 10-14 years old, of different social backgrounds and across the sexes and they found that while many children can solve classroom type mathematical problems, the same children could not apply their knowledge and skills to solve real life mathematical problems. Deboys and Pitt (1988) believe that it is the role of mathematics teachers to ensure that students not only acquire proficiency in basic arithmetical computation but that they should understand the processes they are using and be able to apply them constructively in unfamiliar situations.

Interviewing the expert resulted in the partitioning of an area of syllabus and the focussing on a narrow domain of fraction additions in which to construct an adaptive test (Figure 14). Fractions is an important part of the mathematics curriculum and is a domain commonly used in intelligent tutoring research (for example, (Stern *et al.*, 1996) (Nwana, 1993)) and in adaptive testing (Baumunk and Dowling, 1997). This domain is an important one as it is the basis of many mathematical tasks in the City and Guild curriculum for the current student population. From a student modelling point of view, this domain is small but rich enough to allow all sorts of student behaviour.

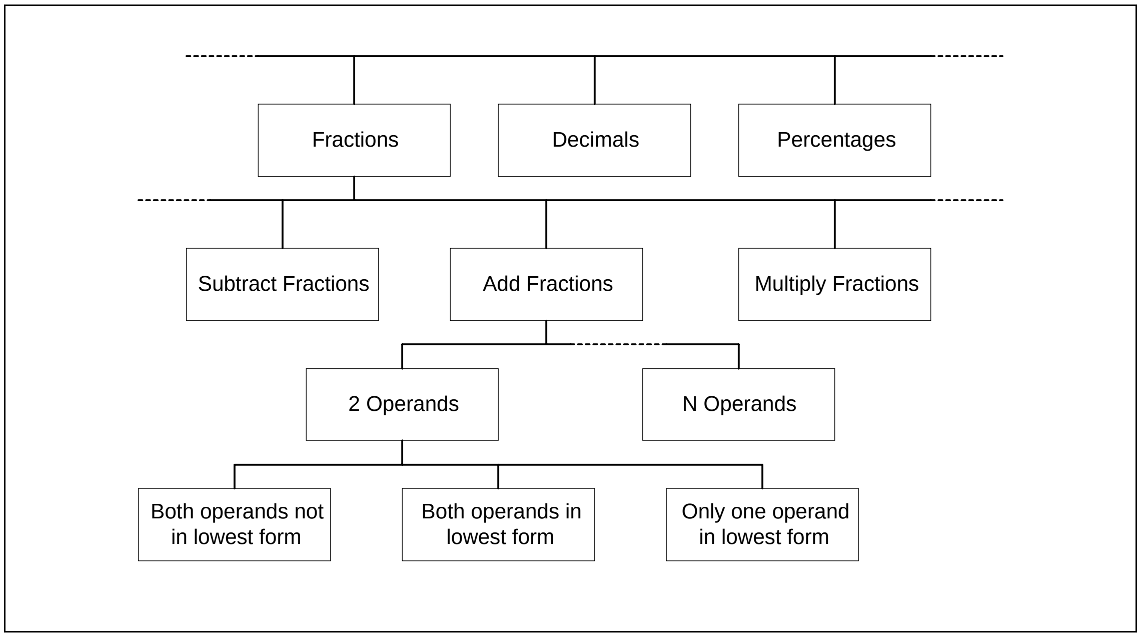

Figure 14. Partitioning an Area of Syllabus

3.2.4 Role of Expert

Due to the transient nature of this type of student population, students enrol at different times. This makes the job of the teacher very difficult especially when there is a need to assess the mathematical knowledge level of each student upon enrolment before appropriate remedial help can be given.

In his current role, the expert provides remedial teaching to an average class size of twelve students. He has four such classes. One of the first tasks of the expert with a new student is to conduct an initial assessment of the student's current knowledge level in mathematics. The transient nature of the student population means different enrolment times which suggest that students are assessed on their mathematical abilities at different times. *Assessment* is one of the most important tasks in teaching and learning as it has a major impact on subsequent remedial help that is rendered (VanLehn and Martin, 1997). By performing one at the start of a course can help the student to overcome isolation and to promote active learning (Taylor, 1998). At present, the initial assessment of the student's state of knowledge in mathematics takes the form of a FIT which is made up of several topics such as fractions, percentages and decimals. A student performs the test in the conventional pencil and paper setting. His answer script is examined by the expert who identifies the areas of weakness. This type of assessment, FIT, has several disadvantages, including the ones discussed in Section 2.5. First, designers of such tests have to ensure that the test is content-balanced, that is, each topic is represented and that no topic or subtopic is over tested or under tested. If a test is not content-balanced, there is a possibility that one or more areas of weakness of the student may not be identified. Secondly, the test is the same for each student and this introduces the possibility of noise, such as copying especially if there is more than one student performing the test at any one time.

After assessment, the students carry out their own remediation with the help of text materials. A book by Llewellyn and Greer (1996) is heavily used. The book is structured such that for each topic, there are subtopics where concepts are explained as expositions followed by a series of problems which are arranged in order of difficulty. Each student works on his respective areas of weakness. The expert advises the students on a *manual adaptive remediation strategy*. The student is to attempt questions of moderate difficulty and if he finds these to be easy, he is to attempt the more difficult ones. Conversely, if he finds them too difficult, he is to attempt the easier ones. This strategy allows the student to further isolate, and at a finer detail, his areas of weakness.

The expert then intervenes with remedial help on a one-on-one basis or in a group if more than one student has difficulty in a similar area. He first explains the underlying concepts before attempting to teach procedural knowledge through a series of problems on the board.

This remediation strategy is *reteaching* and is rather coarse-grained in that it does not focus on specific student misconceptions. The students then attempt to solve similar problems chosen by the expert. In group teaching, students tended to understand at different rates and there is evidence of *peer-to-peer tutoring* where the better students help out their weaker peers.

While there is evidence of a manual adaptive remediation strategy, there is no evidence of adaptive testing, manual or otherwise, as a strategy for assessing a student's knowledge upon enrolment. Although FIT features as the main assessment procedure, adaptive testing is an attractive alternative. The following section presents a small experiment to observe an expert in his one-on-one assessment with students. It was found that the expert conducted a manual form of adaptive testing.

3.2.5 The MATT Experiment

3.2.5.1 Aim of Experiment

The MATT (**M**anual **A**daptive **T**es**T**ing) experiment is experiment was conducted to observe and establish if an adaptive form of testing was adopted by the expert in his one-on-one assessment of the state of knowledge of a student in a subject domain.

3.2.5.2 Subjects

Two school children were invited to participate in the experiment. The two subjects are young children who are currently attending a local school and are following the UK National Curriculum. The first subject is an eleven year old boy at Year 6. The second subject is a ten year old girl at Year 5.

3.2.5.3 Method

The sessions were conducted in a home environment. The chosen domain is fraction addition and subtraction in elementary mathematics. The expert did not use any instructional materials or software tool to perform the test. The expert had access to the problem solving strategy of the students. These interactive sessions were observed by the knowledge engineer and are documented in Appendix B.

3.2.5.4 Findings

The expert was observed to perform a manual form of adaptive testing. The following observations were made.

- **Input bandwidth**

  Human testers have many advantages over computers. The expert could combine data from a wide variety of sources, such as voice effects or facial expressions, an "eureka" look, a puzzled expression, or a hesitant tone of voice (Wenger, 1987) (Holt *et al.*, 1994). In addition, the expert had access to the 'thinking aloud' and problem-solving steps by the student. All these data helped to shape the decision in problem selection and in starting, continuing and stopping the test.

- **Starting the Test with an "Easy" Question**

  The expert was concerned with maintaining the motivation and confidence of the subject at all times and was particularly concerned with the appropriate entry point in which to begin the test. His choice of problems was influenced by the cues he picked from the student. His strategy was to start the test with a problem of lower level of difficulty than the one he thinks the student is capable of solving, that is, with an easier portion of the syllabus. For example, questions were selected differently for each of the two subjects as the first subject exuded confidence in mathematics while the second subject displayed anxiety in the test and a lack of confidence in mathematics in general.

- **Evidence of Redundant Questioning**

  It was observed that the expert adopted a rather loose and ad hoc strategy in his selection of subsequent problems. This is consistent with the findings of Putnam (1987) who observed that teachers use loose curriculum scripts rather than grain assessment. For example, in assessing the knowledge of the first subject, the expert was particularly interested in assessing the mastery level of the subject in a particular skill, *calculate lowest common denominator*, but his choice of a problem, 4/3 + 5/4, did not discriminate against the use of the skill *calculate common denominator* (which could be achieved by multiplying the denominators). There were about three 'redundant' problems before the skill, *calculate lowest common denominator*, was specifically tested out.

- **Stopping the Test**

  It was observed that the expert stopped testing when he found that further questioning would not reveal any new information about the student's knowledge in the subject. In the case of the first subject, all the problems presented had non-common denominators and because the subject showed that he was capable of solving them, the expert did not present him with 'easier' problems such as those with common denominators. Likewise, the second subject was presented with problems of common denominators and because she had difficulty in solving them, she was never presented with 'harder' questions, such as those involving non-common denominators.

- **Diagnostic Strategy**

  This is concerned with the diagnosis of student answers to problems. The expert observed the problem solving steps of each student and modelled the student's knowledge in terms of domain-specific knowledge such as problem solving strategies, and *correct* and *incorrect* knowledge. The cognitive modelling strategy of the perturbation model also maintains the latter type of knowledge where the student's knowledge can be described in terms of correct rules and mal rules. By *correct* knowledge, this means that the expert was looking for a demonstration by the student of the relevant problem solving skills. By *incorrect* knowledge, the expert was looking for common misconceptions; for example, the second subject demonstrated a common misconception where the denominators were added together to give a resultant denominator.

- **Remediation Strategy**

  This is concerned with the remedial help given to the student after the test has taken place. It is observed that the expert did not find it necessary to offer remedial help to the first subject as the student has demonstrated a mastery of the necessary problem solving skills. As for the second subject, remediation took the form of *reteaching* of certain basic concepts of fractions and the problem solving steps of the problems which the student had displayed difficulty in solving.

- **Student Model**

  The type of information which was maintained by the expert about the student was observed to be both learner-related characteristics, such as motivation and confidence, and domain-specific knowledge, such as problem solving strategies, correct knowledge and misconceptions.

3.2.5.5 Experiment Summary

The above experiment confirmed that the expert adopts an adaptive form of testing when assessing student's knowledge in a subject domain on a one-on-one basis. The expert was observed to perform deep cognitive modelling of the student's domain-specific knowledge in terms of problem solving strategies and correct and incorrect knowledge. Despite this, his problem progression strategy for selecting subsequent questions was found to be rather loose, with many redundant questioning. Also, his remediation strategy is coarse-grained reteaching and it did not justify the deep cognitive modelling effort he performed during test administration.

3.3 Conventional Knowledge Acquisition Techniques

Expert emulation involves expressing human knowledge and strategies in a computer system and is often referred to as the *bottleneck* problem (Murray, 2000). Knowledge acquisition is a difficult and time-consuming process which involves many hours of interaction between the expert and the knowledge engineer. There is also the problem of choosing the right tool for eliciting the appropriate knowledge type.

A knowledge engineer has a choice over many knowledge acquisition techniques, such as concept analysis, unstructured interviewing, structured interviewing, domain and task analysis, process tracing and protocol analysis and simulations and automated tools (McGraw and Harbison-Briggs, 1989). The choice of one technique over another depends on the type of knowledge which the knowledge engineer wishes to elicit from the expert. There is a mapping of types of knowledge to knowledge acquisition techniques (Table 1). For example, if the knowledge acquisition activity for a particular phase is to “identify general heuristics that are available on a conscious level”, the knowledge engineer would be seeking knowledge that is primarily declarative in nature. Declarative knowledge is generally available in the short-term memory, which allows the domain expert to express it verbally.

The choice of technique for this study is likely to be structured interviewing and task analysis as the types of knowledge to be elicited are likely to be declarative (domain knowledge and student model) and procedural (problem progression strategy).

Knowledge	**Activity**	**Suggested Technique**
Declarative knowledge	Identifying general (conscious) heuristics	Interviews
Procedural Knowledge	Identifying routine procedures/tasks	Structured Interview Process Tracing Simulations
Semantic Knowledge	Identifying major concepts/vocabulary	Repertory Grid Concept Sorting
Semantic Knowledge	Identifying decision making procedures and heuristics (unconscious)	Task Analysis Process Tracing
Episodic Knowledge	Identifying analogical problem solving heuristics	Simulations Process Tracing

Table 1. Correlating Knowledge Type and Acquisition Technique
(McGraw and Harbison-Briggs, 1989)

3.4 Constraint Logic Programming

3.4.1 Background of Constraint Logic Programming

Constraint Logic Programming, or CLP for short, has been heralded by ACM (Association for Computing Machinery) as one of the strategic directions in computing research (Marriott and Stuckey, 1998). CLP provides a language for the description of relationships in the form of constraints and a mechanism to calculate a set of values which satisfy those constraints. Constraint programs are often written to provide *optimal* solutions to problems. CLP comes under the paradigm of *Constraint Solving* which is a powerful paradigm that allows a natural representation of complex problems (Lassez, 1987).

Constraint Logic Programming is an extension of Logic Programming aimed at replacing the pattern matching mechanism of unification, as used in Prolog, by a more general operation called constraint satisfaction or constraint solving (Cohen, 1990) (Constraint Programming, 2000). Logic Programming is a language paradigm based on logic. It shot to fame via the Prolog language as a consequence of the Japanese Fifth Generation project and the expert systems boom of the mid 1980s (Pountain, 1995). Logic programming is characterised by two components: *resolution* and *unification*. *Resolution* is an inference step used to prove the validity of predicate calculus formulas expressed as clauses while *unification* is the matching of terms used in a resolution step (Cohen, 1996). Prolog is based on first-order predicate logic and the objects that it manipulates are pure symbols with no intrinsic meaning. Execution of Prolog program proceeds by searching a database of such facts to find those values that will satisfy a user's query, using a process called *unification* based on syntactic identify. Since Prolog tries to find the set of all solutions to a query, during this search many dead-ends may get explored and then abandoned by backtracking to an earlier state and trying a different branch. For complex problems, this search process can take up both space and time, which can lead to inefficiency.

Although a relatively young paradigm, the use of CLP in industry since its inception in the late 1980s has resulted in many successful real-life applications. CLP is especially well suited to solving problems in scheduling. Examples include container port scheduling (Abbott, 1995) and nurse scheduling (Darmoni *et al.*, 2000). The history and background of constraint programming is usefully summarised by Marriott and Stuckey (1998). Constraint programming modules are available for a range of programming platforms. Examples include Prolog II and Prolog III (Jaffar and Lassez, 1987), CHIP, clp(R) and clp(FD).

Constraint Logic Programming provides a language for the description of relationships in the form of constraints and a mechanism to calculate a set of values which satisfy those constraints. The basic components of a problem are stated as *constraints* while the problem as a whole is represented by putting the various constraints together as *rules*. A problem is defined in terms of its variables and in terms of the constraints that must be solved by these variables. Two types of constraints exist – *domain constraints* and *relational constraints*. *Domain constraints* refer to constraints on the range of values each variable can take while *relational constraints* refer to constraints placed on the relationship between the variables. Solutions to such problems can be found by using the constraints to detect impossible combination of values and arriving at an *optimum* solution.

CLP has been particularly geared to solving *Constraint Satisfaction Problems*, or CSPs. A *Constraint Satisfaction Problem* consists of a set of variables, each of which has a discrete and finite set of possible values, and a set of constraints between the variable which specify which combinations of values are allowed and which are not. Variables may have integer or symbolic domains. A solution to a CSP is a set of variable-value assignments which satisfies all the constraints. It involves network consistency check algorithms (Tsang, 1993), constraint propagation, and backtracking search. In essence, the algorithms increase the efficiency of the search by *looking ahead,* and *actively* using the constraints to prune the search space, thus minimising backtracking. Optimisation is based on a form of *branch and bound,* that is, as soon as a solution is found, a further constraint is added to the effect that the value of the optimising criterion must be less than the value just found. This causes the system to backtrack until a better solution is found. When no further solution can be found the optimum value is known.

A significant advantage of CLP over the standard implementation of Prolog is that CLP can perform arithmetic with uninstantiated variables. As a simple example, consider this code fragment which can be used to calculate the integer side lengths of right angle triangles:

```
domain([X,Y,Z], 1, 99),
Z*Z  #= X*X + Y*Y,
X #< Y,
labeling([], [X,Y,Z]).
```

The first line introduces a list of variables, and specifies that they may take values in the range between 1 and 99. The second line presents the Pythagorean constraint. The "#" symbol is used to indicate a relational constraint. The third line constrains X to be less than Y. This usefully eliminates solutions which differ only in the ordering of X and Y; for example, we do not need both 3,4,5 and 4,3,5 as solutions. The final line initiates a search for solutions for X, Y and Z. The way this search is carried out is controlled by the first argument of "labeling". This is the *constrain and generate* methodology: first, the constraints are applied, then a solution is generated by labeling. The empty list symbol, [], indicates that the default strategy is to be used. A backtracking search is used, which explores the domains in ascending order. When compiled and run, this code will provide values for all three variables, X, Y, and Z. Given that all these input variables were uninstantiated at the start, Prolog would not have been able to produce any instantiations unless only one variable was not instantiated.

3.4.2 Constraint Logic Programming as a Tool for Knowledge Acquisition

A type of constraint logic programming, clp(FD), for knowledge acquisition is used in this study. Clp(FD) or constraint logic programming over finite domains (Carlsson *et al.*, 1997) is a particular implementation of constraint logic programming that is integrated with SICStus Prolog, a commercially available Prolog system developed and distributed by the Swedish Institute of Computer Science (SICStus, 2002). This implementation is used for the practical work of this thesis.

As clp(FD) is particularly suited to representing constraint problems with a finite number of discrete solutions. It has a range of built-in procedures for search for optimal solutions. It also has applications in knowledge acquisition. Knowledge acquisition is the process of acquiring knowledge from human experts which is entered into a computer and organised for use in an expert system. It is essentially made up of two processes – knowledge elicitation and knowledge representation. Clp(FD) is used to represent classes of arithmetic problems, and it is also used in the knowledge elicitation process.

Clp(FD) may be actively used by the interviewer when conducting knowledge elicitation interviews. The teacher, who is the target of the emulation, is not expected to write constraints, though is more than likely to take an interest in them. During discussions, which involve the production of example problems, the interviewer enters the necessary constraints, or modifies existing constraints, to describe the particular class of problem under discussion. The set of constraints is then solved interactively to produce example problems. These allow the interviewer to obtain confirmation of what had been elicited and form the basis of further rounds of discussion and modification. For most classes of problems, it is not feasible to expect the teacher to inspect every example. This means that unexpected and undesirable examples may be not be revealed during this knowledge elicitation process. Traditional program testing and additional knowledge elicitation sessions are needed to reduce the probability of errors.

Using clp(FD), domains can be enlarged or restricted, and constraints can be added or removed. It is easy to represent both the mathematical structures of problems and it is also easy to control the choice of numeric values incorporated in problems. As with all knowledge elicitation, good preparation by the interviewer is extremely valuable. This can conveniently take the form of developing some speculative constraints, but this should not be allowed to influence the interviews. The aim of emulating the expert must be paramount. Although the technique is used here for simple arithmetic problems, it could also be applied to a much wider range of problems involving the manipulation of a finite number of categories.

The following section discusses the elicitation of the domain knowledge and the role of clp(FD) in knowledge elicitation, knowledge representation, problem generation and answer evaluation.

3.5 Domain Knowledge Representation

Knowledge representation is the task of writing down, in some language or communicative medium descriptions or pictures that correspond in some salient way to the world or a state of the world. In Artificial Intelligence, this is concerned with writing down descriptions of the world in which an intelligent machine might be embedded in such a way that the machine can come to new conclusions about its world by manipulating these symbolic representations (Levesque, 1986). Techniques such as semantic networks, frames, and rules have been used for this purpose. For many ITSs, a body of knowledge has been described in terms of skills, concepts and problems. Examples include a granularity hierarchy (McCalla *et al.*, 1992), a curriculum hierarchy (Huang, 1996), a curriculum tree (Chan, 1992), a topic network (Beck *et al.*, 1997) and a skills graph (Mao and Lin, 1992). For many CAT systems, a test syllabus has been described in terms of problems (such as IRT systems) or classes of problems (such as KST systems).

3.6 Eliciting the Domain Knowledge

Having decided on the area of syllabus to be tested, for example fraction additions in this study, the next step is to elicit the strategy of the expert in describing the component or basic item of the domain knowledge. Through interviewing, it was revealed that the expert's approach involves an exhaustive review of all classes of problems within a test syllabus. This indirectly handles the task of content-balancing, commonly associated with IRT-based systems, as an exhaustive declaration of problem types implies that no area within the domain is under or over tested. Clp(FD) was used by the interviewer when conducting knowledge acquisition interviews. For example, after discussing example problems with the expert, the interviewer used clp(FD) to represent these problems and to generate as many example problems. The result is a domain of problems categorised according to their features and response types. This is discussed in following sections.

3.6.1 Categorising Problems

There are different ways of categorising problems. For example, studies have shown that competent or expert problem solvers could readily categorise word problems and they tend to categorise problems differently from novices and this is because experts hold a *richer* body of knowledge about the subject matter (Chi *et al.*, 1981). In this study, the expert has chosen to categorise problems into several classes according to features such as common or non-common denominators and the range of possible values of numerators and denominators, based on the expression:

$$N1/D1 + N2/D2 = N/D$$

This strategy in categorising problems can be represented by constraints. Clp(FD) provides a *declarative* and *executable* means of describing such specifications, and can be made sufficiently convenient for it to be used *on the fly* during a knowledge elicitation session involving the expert and the knowledge engineer. Such software facilitates the capture of descriptions of classes of problems and also descriptions of possible responses of a student to those problems. These executable descriptions can be used to generate examples which can form the basis of several rounds of discussion between the expert and the knowledge engineer. An example is given in the following code fragment which was used during knowledge elicitation.

```
?- use_module(library(clpfd)).

generate_problems(N1,D1,N2,D2,N,D):-
    domain([N1, N2], 1,8),
    domain([D1, D2], 2,9),
    domain([N,D],1,10),
    labeling([], [N1,D1,N2,D2,N,D]),
    N1/D1 + N2/D2 =:= N/D.
```

The first three clauses of the *generate_problems* predicate are *domain* constraints which dictate the range of possible integer values which can be taken by any variable. Constraint solving is achieved by the *labeling* predicate which will initiate a search for solutions for all the variables in the arithmetic expression N1/D1 + N2/D2 = N/D.

A solution from the execution of the above code is:

$$N1 = 1, D1 = 2, N2 = 1, D2 = 2, N = 2, D = 2$$

or $\frac{1}{2}+\frac{1}{2}=\frac{2}{2}$, which satisfies the constraints. More solutions or problem instances can be generated, for example, $\frac{1}{2}+\frac{1}{4}=\frac{3}{4}$ and $\frac{1}{2}+\frac{1}{5}=\frac{7}{10}$.

The expert segregated these different instantiations of problems into classes of problems. The result, as shown in Figure 15, is a declaration of an exhaustive list of problem classes which represents the different configurations of problems in the domain of fraction addition of two operands. The clp(FD) representation of the problem classes is given in Appendix C.

PT1: Add two proper fractions with common denominators
PT2: Add two improper fractions with common denominators
PT3: Add a proper fraction and an improper fraction with common denominators
PT4: Add two proper fractions of different denominators which are multiples of one another
PT5: Add two improper fractions of different denominators which are multiples of one another
PT6: Add a proper and an improper fraction of different denominators and are multiples of one another
PT7: Add two proper fractions of different denominators which are not multiples of one another
PT8: Add two improper fractions of different denominators which are not multiples of one another
PT9: Add a proper fraction and an improper fraction of different denominators and are not multiples

Figure 15. Classes of Problems

The use of constraints for describing classes of problems is similar to the efforts by Hirashima *et al* (1996) who constructed a pool of Physics word problems - equivalent problems, partial problems and specialised problems - by eliciting from human tutors and practice materials. They did not, however, mention the use of any specific constraint language. The use of constraints in this study can also be compared with that of Ohlsson's constraint-based student modelling technique (Ohlsson, 1994). Like our approach, Ohlsson represented domain knowledge as a set of constraints, but while our approach uses constraints to describe the explicit features of each problem class, Ohlsson uses constraints to detect erroneous student answers. He uses a representational format called *state constraints.* A state constraint is an ordered pair **$<C_r,C_s>$** where C_r, the *relevance condition*, identifies the class of problem states for which the constraint is relevant, and C_s, the *satisfaction condition,* identifies the class of (relevant) states in which the constraint is satisfied. Each member of the pair can be thought of as a conjunction of features or properties of a problem state. Consider the problem of adding two fractions. For example, the idea that fractions have to have equal denominators before they can be added can be expressed in state constraint form as:

if the problem is n1/d1 + n2/d2 and if n = n1 + n2,
then it had better be the case that d1 = d2 or else something is wrong

The relevance conditions are $n1/d1 + n2/d2$ and $n = n1 + n2$. The first condition is relevant only when one is adding fractions. The second condition is relevant only when the denominators are equal. The satisfaction condition, $d1 = d2$, is satisfied only if the relevance conditions are true. State constraints are elicited from experts and captured the central concepts of the domain. Any violated constraints represent the student's erroneous behaviour and are used to guide subsequent remediation.

Another difference is that Ohlsson's approach is mainly declarative such that the domain and student model are not executable. Our approach goes a step further by using constraint solving, thus making the domain and student model executable; the advantages of this strategy are discussed in Section 3.6.3 and Chapter 4 respectively.

3.6.2 Categorising Responses

The expert identified a list of answer or response types that are possible with the current domain, as shown below:

RT1:	Proper fraction in its simplest form (e.g. 1/2)
RT2:	Whole number = 1 (e.g. 3/3)
RT3:	Proper fraction which can be simplified further (e.g. 6/8)
RT4:	Improper fraction in its simplest form (e.g. 4/3)
RT5:	Improper fraction which can be simplified further (e.g. 10/6)
RT6:	Whole number > 1 (e.g. 8/2)

Figure 16. Types of Possible Responses

It was found through *rapid prototyping* that not all response types could be generated for each problem class of Figure 15. For example, the problem class "Add two improper fractions with common denominators" could never yield a proper fraction as a response type. A list of possible response types is given in Figure 16. The set of problem classes can be further classified according to their possible response types. This is given in Appendix D.

3.6.3 Domain Representation in clp(FD)

Once the description of a class of problems and their appropriate responses is treated as a set of constraints – domain and relational – it must be satisfied by every example of that class and this is achieved through constraint solving. Classes of problems with possible response types make up the problem domain as *constraint logic programs.* Each problem class consists of a set of variables, a statement of the *domain constraints* that determine the range of integer values that each variable can hold, and a statement of the *relational constraints* that hold between the variables. For example, if the expert wanted to represent a class of problems of type PT1 and he wanted to use single digit integers, this can be represented as the following code fragment in clp(FD):

```
domain([N1, N2], 1, 8),          % Single digit integers
domain([D1, D2], 2, 9),
N1 #<  D1,                       % First operand - proper fraction
N2 #<  D2,                       % Second operand - proper fraction
D1 #= D2.                        % A common denominator
```

The expert can also specify a response type. For example, he may want to specify a response type as a proper fraction, that is, RT1 (see Figure 16). This can be achieved through the constraint:

N #< D

where N and D can be specified to take any integer value from 1 to 99. These constraints can be added to the previous code fragment, thus:

```
domain([N1, N2], 1, 8),          % Single digit integers
domain([D1, D2], 2, 9),
domain([N,D], 1, 99),            % Possible values for the answer
N1 #<  D1,                       % First operand - proper fraction
N2 #<  D2,                       % Second operand - proper fraction
D1 #= D2,                        % A common denominator
N #< D.                          % Answer must be a proper fraction
```

Likewise, if the intention was to have a result that is an improper fraction, the constraint N#<D can be replaced by N #> D.

As a more complete example, consider the following interview excerpt:

> Expert: If I have a student who displays a dislike for mathematics and little confidence in general, I would want to ensure that I do not start with a difficult question. I would use single-digit integers *(domain constraint)* and start the test with a simple problem which involves the addition of two proper fractions of a common denominator, in their lowest form, which would yield another proper fraction in its lowest form. This way, only one skill is needed, that is, the student needs only to add the fractions without having to bring it to its lowest form. For example, I would give him 1/3 + 1/3 and not 1/8 +1/8."

From this excerpt, the knowledge engineer first identifies the domain and relational constraints. A class of problems which satisfy these constraints can be built and be represented in clp(FD), as shown in Figure 17. A solution from the execution of the code is:

N1 = 1, D1 = 3, N2 = 1, D2 = 3, N = 2, D = 3

which can be formatted as $\frac{1}{3}+\frac{1}{3}=\frac{2}{3}$, which satisfies the constraints. More problem instances can be generated, for example, $\frac{1}{5}+\frac{2}{5}=\frac{3}{5}$ and $\frac{1}{9}+\frac{4}{9}=\frac{5}{9}$.

The problem class represented in Figure 17 corresponds to problem type PT1_RT1 given in Appendix D which is the addition of two proper fractions with common denominators with a response type of a proper fraction in its simplest form. Each problem class given in Appendix D can be represented as a constraint logic program. The use of the clp(FD) formalism in this way gives rise to an *executable* problem domain which facilitates problem generation and the evaluation of student answers. Unlike many student modelling systems, these two routines do not need to be coded separately; they are discussed in the following subsections.

```
?- use_module(library(clpfd)).
qt(N1,D1,N2,D2,N,D):-
  domain([N1,D1,N2,D2], 1,9),    % Single digit integer numerators
  domain([N,D],1,99),            % Possible values for answer
  N1 #< D1,                      % First operand - proper fraction
  N2 #< D2,                       % Second operand - proper fraction
  D1 #= D2,                       % A common denominator
  D #= D1,                        % Same denominator in solution
  N #< D,                                  % Answer is a proper fraction
  labeling([],[N1,D1,N2,D2, N,D]),        %Generate values
  \+ cancel(N1,D1,_,_),           % First operand in lowest form
  \+ cancel(N2,D2,_,_),           % Second operand in lowest form
  \+ cancel(N,D,_,_),             % Answer in lowest form
  N1/D1 + N2/D2 =:= N/D.         % Arithmetic expression
%
% Skill:  Cancel fraction e.g. N/D into lowest form X/Y e.g. 63/81 gives 7/9
cancel(N,D,X,Y) :-
  domain([N,D,X,Y,F], 1,99),     % F is the highest common factor
  F*X #= N,
  F*Y #= D,
  maximize(labeling([], [F,X,Y]), F),
  F \== 1.                        % To ensure cancel fraction has taken place
```

Figure 17. A Problem Class and a corresponding Response Type

3.6.3.1 Problem Generation

Problems can be generated *on the fly* by executing a problem class. This is achieved through constraint solving, as illustrated in the example in Figure 17. An advantage is that, unlike IRT systems, there is no need to maintain huge repositories of test items or problems.

There are two uses of this facility – during knowledge acquisition and during the delivery of the adaptive test. During knowledge acquisition which involves the production of example problems, the knowledge engineer enters the necessary constraints, or modifies existing ones, to describe the particular class of problem under discussion. The set of constraints is then solved interactively to generate example problems. During the delivery of an adaptive test, one or more problems can be generated from a problem class and response type. Although constraint programs are often written to provide optimal solutions to problems, their use can be to generate multiple solutions, in this case, to generate more than one problem for a specific problem class. In adaptive testing, this has the advantage of ensuring that no two tests look identical thus reducing the possibility of copying.

3.6.3.2 Evaluating Student Answers

The use of clp(FD) has another advantage when a diagnostic approach is required to evaluate student answers to problems. It allows a "shallow" type of diagnosis where a student's final answer is evaluated as *correct* or *incorrect.* When a problem is generated, an answer to the problem is also generated. During diagnosis, this answer is checked against that of the student. This type of diagnosis will provide information on what problems have been successfully or unsuccessfully attempted. It differs from deep cognitive modelling in that the student's solution path or final answer is not inspected to reveal detailed domain-specific information such as the student's problem solving strategy and misconceptions.

3.7 Eliciting Other Information

Other pieces of information were also elicited from the expert. These are the identification of problem solving skills for the current domain and the expert's strategy in measuring problem difficulty.

3.7.1 Categorising Problem Solving Skills

The expert identified the following problem solving skills which are commonly used in solving problems in the current domain.

- Add equivalent fractions
- Cancel fraction
- Make proper
- Find the lowest common multiple
- Find equivalent fractions

Multiple solution paths may exist for any problem and the skills used in one solution path may differ from those used in another solution path. Also, students usually devise their own problem solving strategy and may not use all the skills expected to be applied to solve a problem. An example is given in the MATT experiment described previously where the first subject was asked to solve ¾ + ½. The expert had expected the student to use the skills *find common denominator, find equivalent fraction* and *make proper,* but instead the student used the skills *number facts, sum whole numbers* and *add equivalent fraction.*

Each of the problem solving skills listed above could be represented in clp(FD). For example, the *cancel fraction* skill can be represented in the following clp(FD) code fragment.

```
% Simplify the fraction N/D into its lowest form to give X/Y
        % Example:  63/81 gives 7/9
        cancel_fraction(N,D,X,Y) :-
                domain([N,D,X,Y,F], 1,99),
                F*X #= N,
                F*Y #= D,
                maximize(labeling([], F,X,Y]), F).
```

However, the use of clp(FD) here is optional and the skill can be coded in any other procedural or declarative language such as Prolog. This is because an optimal use of constraint logic programming would be when none of variables are instantiated. The *cancel_fraction* predicate would normally be invoked with the input parameters, N and D, already instantiated to integer values. The variable F is the common factor to be cancelled. This is specified by the two relational constraints. The final line initiates a search for solutions for X, Y and F. The way this search is carried out is controlled by the first argument of labeling. The empty list symbol, [], indicates that the default strategy is to be used, that is a backtracking search which explores the domains in ascending order. The *maximize* predicate in the final line ensures that the largest value of F will be found.

3.7.2 Measuring Problem Difficulty

One of the main advantages of adaptive testing over fixed-item testing is the significant reduction in the test length, and consequently, in the testing time (Ríos *et al.*, 1999). This is facilitated by the ability to present a problem of appropriate difficulty to the student at the right time. Problem difficulty can be measured at either one of two stages – *prior to* or *during* test delivery.

The measure of problem difficulty *prior* to test delivery is perhaps the more common approach. A simple strategy is to rank the difficulty of a problem as proportional to the number of identifiable skills required to solve it (Beck *et al.*1997). The skills are not ranked in order of difficulty. Lee (1996) highlighted that this strategy of measuring problem difficulty by the number of skills is just one of the many factors. Others that might be used include:

- *Number of steps required to solve the problem.* This may differ from the number of skills required to solve the problem as a skill may be used more than once.
- *Student's familiarity with the problem.* If a student recognises a problem (Davis, 1984), he is more likely to be able to solve it correctly even if the problem is considered a difficult one. This is especially relevant in situations where tutoring has been conducted based on past examples (Renkl, 1997; Ross and Kennedy, 1990; Chi *et al.*, 1989).

The measure of problem difficulty can also be captured empirically. Response time can be used to calculate the total elapsed time between problem presentation and response. The rationale behind this is that difficult questions require more processing time to solve. Lee (1996) identified an *item difficulty ratio* which is a ratio of the number of respondents answering correctly to the total number of responses to the problem. Another strategy is to *calibrate* a set of problems for a population of students such that a problem is considered *easy* if a high percentage of the population can solve it, or *difficult* if only a small percentage can solve it. This is the approach undertaken by IRT designers. However, this is based on a rather straightforward assumption that a question answered correctly is easy while that answered wrongly is difficult.

Some argued that the measure should indicate how much *cognitive effort* is required from the student (Lee, 1996). This is because a problem may be considered difficult due to it being *poorly phrased* or *misleading*, rather than requiring the student to understand difficult concepts or perform complex calculations. Once the problems is rephrased or clarified, it may be easily solved. On the other hand, difficult problems might be answered correctly for reasons other than thoughtful replies; for example, there may be clues in the question which point to the correct answer. An interesting observation by Beck *et al.* (1997) is that some students rate the difficulty of problems on surface features. For example, they may find problems with single-digit integers easier to solve than those with two or more digits, although these problems may require the same set of problem solving skills. For instance, students found the problem 2/3 + 4/3 easier to solve than the problem 17/18 + 19/18 although both these problems require the application of the same skill.

The strategy of measuring problem difficulty *prior* to test delivery leads to the creation of a database of problems, each attached with a measure of difficulty. The expert in this study employs this strategy and is similar to that of Beck (1997) where each problem is attached a difficulty level proportional to the number of skills needed to be applied.

A less common but effective approach is to measure problem difficulty *during* test administration. This approach supports the notion that different students may rate the difficulty of problems differently. Examples include SIETTE (Ríos *et al.*, 1999) and CBAT-2 (Huang 1996). Certain item parameters are identified, such as the number of times a question has been posed to the student, and, the correctness of a student's response to the present question. These item parameter values are updated after every student's response to a question and are used to recalculate the student's new proficiency level and the difficulty level of each remaining question in the database. The system updates the temporary student model and uses this information to guide the test. Once the test terminates, the temporary student model becomes the student's current knowledge.

3.8 Conclusion

This chapter first provided a foundation in which to carry out knowledge acquisition. It introduced the problem domain of elementary arithmetic with fraction additions and has discussed the teaching background of the teacher who is the "expert" in the expert systems aspect of the thesis. It also described an observational study of the expert's assessment techniques which paid particular attention to the issue of "adaptability".

The chapter then showed how knowledge acquisition can be supported by software to produce both a *declarative* description of classes of problems and an *executable* procedure which can evaluate student answers and produce sample problems. These samples can be used as part of the knowledge acquisition process to refine the representation of a problem domain, and they can also be used in the ultimate delivery of an adaptive test. The chapter discussed the use of software support for describing problem solving skills which may be used to categorise problems. This ability is relevant to the task of estimating the difficulty of a problem and this, in turn, is relevant to the sequence in which problems are presented during tests.

Chapter 4.

Initial Experiments: Creating a Student Model and Problem Progression in Adaptive Testing

4.1 Introduction

This chapter discusses the experiments in creating a student model and a problem progression strategy in SKATE. It first discusses the usefulness of a student model for adaptive testing and seeks to determine the contents of such a model. It bases its decision on the findings of two experiments, such as the remediation strategy of the expert after testing and concludes with the choice to maintain domain-specific information about the student, in terms of what is believed to be mastered and a record of successful and unsuccessful attempts at problems. A clp(FD) representation means that the overlay student model is executable and is useful for predicting a student's performance and for generating problems during remediation.

The chapter next presents two distinct strategies for problem progression based on expert emulation. The first experiment is the development of a computer-aided procedure to systematically query an expert to extract a test item sequence called BT. The second experiment describes a knowledge elicitation exercise which captures the expert's testing strategy called XP, which is based on his measure of problem difficulty by the number of skills needed to solve a problem.

4.2 The Use of a Student Model

A student model may contain domain-specific information and learner-specific characteristics about a student. As mentioned in Chapter 2, the former type of information is still more commonly maintained in a student model mainly because of the difficulty with capturing the latter type of information. In adaptive testing, the type of information that is maintained about the student is also mainly domain-specific and such information is used to select the next problem, or to determine when to stop the test, or to guide subsequent remediation. For example, IRT systems maintains information about a student as a single proficiency estimate together with a confidence level while KST systems maintains student information as a knowledge state which is a set of problems, problem classes or skills which the student has displayed mastery of.

Domain-specific information about the student is commonly captured through an analysis of student responses to problems (Brusilovskiy, 1994). This may be achieved in different ways:

a. Characterising final answers

This is the simplest form of diagnosis which involves the characterising of a student's final answer to a problem as correct or incorrect, without inspecting the answer in detail or accessing the intermediate steps of the student. There is a trade-off between accuracy and speed and it is the most common diagnostic strategy of adaptive testing systems. An answer is treated as incorrect even if it is, upon closer inspection, partially correct. The measure of mastery of the skills associated with the problem can be increased for a correct response, or decreased for an incorrect response. If an ITS employs a perturbation model, then the cause of an incorrect response can be determined by a perturbation in either the student model or the buggy library. The advantage of this strategy is its robustness in the face of bug migration and radical strategy variability as it is not affected by inconsistent behaviour or multiple problem solving strategies of the student.

b. Analysing problem solving steps

Analysing a student's problem solving steps or solution paths provides additional information about the student's cognitive processes and avoids having to infer from large search spaces or deal with the problem of combinatorial explosion. The task is to observe the student's problem solving steps of *known* problems, either online or offline. The analysis of the student's actions of unknown problems is not discussed here but is the most complex form of analysis; an example is WEST (Burton and Brown, 1985) which uses the differential modelling technique. The analysis of solution paths of known problems is usually made possible because an ITS usually maintains procedural knowledge. Examples include GUIDON (Clancey, 1979) which observes medical students during problem solving, and the LISP Tutor (Anderson and Reiser, 1985) and FITS (Nwana, 1993), both of which maintain a set of correct procedures as well as buggy rules to monitor students during problem solving by the model tracing technique.

c. Deducing from final answers

This involves deducing a student's solution path from a final answer. For example, if there is an executable student model as an overlay model and a buggy model, it can be used to simulate the problem solving steps of the student, from the final answer by using a set of correct and incorrect knowledge elements to entire procedures. Examples include DEBUGGY (Brown and Burton, 1978), PROUST (Soloway and Johnson, 1984) and ACM (Langley and Ohlsson, 1984). However, there is no guarantee that the inferred solution path represents the true problem solving strategy of the student, as there is the possibility of idiosyncratic behaviour (Nwana, 1993) and radical strategy variability (Ohlsson, 1994) on the part of the student.

The first type of diagnosis reveals domain-specific information in terms of existing knowledge and gaps in knowledge while the second and third types capture information in terms of problem solving strategies and misconceptions. The following subsection describes an experiment carried out to diagnose student answers to problems in two modes – characterising final answers and analysing problem solving steps. The diagnosis was carried out manually by the expert. The findings reveal that the second mode provides more detailed domain-specific information about the student than the first mode.

4.2.1 The DSA Experiment

4.2.1.1 Aim of Experiment

The aim of the DSA (**D**iagnosing **S**tudent **A**nswers) experiment is to compare two different modes of diagnosis. Both modes yield domain-specific information about each student in a subject domain. The first mode is a simple form of diagnosis which inspects final answers and characterises each answer as correct or incorrect. The second mode performs a detailed diagnosis or deep cognitive modelling of the student answers by examining all the working or problem solving steps leading up to final answers.

4.2.1.2 Subjects

The test was presented to twelve inmates who are part of the student population described in Section 3.2.2.

4.2.1.3 Method

A pencil-and-paper fixed-item test was presented to the subjects; a copy of the test is given in Appendix E. It covers the topic of fraction additions and consists of thirty-five questions which represent all the problem classes listed in Appendix D. The questions are arranged in an order of difficulty: solve a problem requiring a basic skill such as calculate lowest common denominator, add two fractions with common denominators, add two fractions with different denominators. Table 4 in Appendix F shows each question corresponding to a problem class and a set of problem solving skills which can be applied to solve the question. The first eleven questions asked specific basic skills, such as *cancel fraction*, and do not correspond to any problem class declared in Appendix D.

No time limit was imposed on completing the test and the students were not allowed to confer with one another. They were instructed to write down their intermediate problem solving steps on the answer scripts. The test was invigilated by the expert, who is their teacher, in a formal classroom setting. The answer scripts were collected after the test and inspected by the expert in the two modes of diagnosis.

4.2.1.4 Findings

The results of the two modes of diagnosis are given in Table 5 and Table 6 of Appendix F. Both modes reveal the evidence of "downward consistency" which is a term coined to mean that if a student has successfully answered questions which have required more than one problem solving skill, say {a,b,d}, then it was observed that he could also successfully answer questions which required fewer but similar skills, say {a} or {a,b}. For example, the students who have answered question 5 correctly (which required skills *a* and *b*) have also answered question 1 correctly (which required skill *a*). "Downward consistency" was also observed by Marshall (1981). It has an important implication in that the number of candidate problems to be selected can be reduced.

The second mode reveals more detailed domain-specific information than the first mode, such as noise and misconceptions. These are discussed below.

- **A correct final answer may be due to copying**

Although the students were instructed to display their problem solving steps for each question, there were varying degrees of working shown. Some presented full workings while others presented either partial workings with some missing steps or no working at all. If a correct final answer to a problem was accompanied by full working, this implies that the student has successfully solved the problem and that he has mastered the relevant skills needed for that problem. When no working is given, a correct final answer may imply that the student has copied. There is therefore a possibility of *noisy* data caused by copying. Evidence of this possibility can be found in the answers of subject I for questions 24 to 35 where the subject did not show any working to any of these problems but provided a correct final answer for each of them. One of the skills required in each of these questions was *calculate lowest common denominator*. If "downward consistency" was true, then the subject should be able to answer some earlier but easier problems (questions 6 to 9) which have required the use of only one skill, that is, *calculate lowest common denominator*. However, when the subject did not successfully solve these easier problems, it implied that he had copied.

- **A correct final answer may be due to guesswork**

A possibility of a final answer being categorised as correct but is in fact incorrect may be caused by another type of noise, that is, guesswork. There is no evidence of this type of noise in this experiment, but is worth mentioning here.

- **A correct final answer may be due to a misconception or stable error**

Another possibility that is not evident in this experiment but is also worth mentioning is a situation where a correct answer may be caused by a misconception or *stable error* (VanLehn, 1990). An example is provided by Borasi (1994) where a student was asked to solve 16/64 and he provided the correct final answer as ¼. No working was provided by the student but a correct response implies that the student has successfully used the skill, *cancel fraction*. However, when the student was asked to solve another problem 18/84 and he provided the answer 1/4, it became clear that he did not use the skill *cancel fraction* and this drew suspicion to the earlier problem of 16/64, which the student had 'successfully' solved. It then became obvious that the student had applied a misconception by cancelling a common digit from the numerator and denominator. This influenced subsequent remedial help which dealt at a more basic level, that is, at the concept of a fraction, rather than providing help with the skill, *cancel fraction*.

- **A wrong final answer may be partially correct**

A closer inspection of the final answers revealed that some were partially correct. For example, the answers to question 5 of subject E and to question 19 of subject G were found to be partially correct and it was evident that they did not apply the skill *cancel fraction*. This implies a lack of mastery of this particular skill. Another example can be found in the answers to question 17 and 20 of subject I where he did not apply the required skill *make proper*.

- **A wrong final answer may be due to misunderstanding of question**

There is also a possibility of a wrong answer being caused by a misunderstanding of the question. This might have been the case with the answers to questions 10 and 11 by subject D and this may be due to a poorly phrased question, as discussed earlier by Lee (1996). Although subsequent remedial help may focus on the skills required to solve these problems, it may be the case where the questions needed to be rephrased or clarified for them to be easily solved.

- **A wrong final answer may be due to careless slips**

It is not difficult for a teacher to detect a careless slip from other types of noisy data such as guesswork or copying as this inspection usually takes place in relation to the student's answers to previous problems. Examples of careless slips can be found in the answers to questions 29 and 33 by subjects F and J respectively. These subjects can be classified as 'good' students as they have successfully solved the majority of the test questions. A possible reason for careless slips may be that the students had to wade through the easier problems before reaching the more difficult ones, and slips may occur due to boredom or demotivation. Another possible cause may be *forgetting* due to *cognitive overload* which may interfere with the process of problem solving and lead to errors (Sweller, 1988) (Kashihara *et al.*, 1994).

- **A wrong final answer may belong to a question which was not attempted at all**

A blank answer is usually characterised as incorrect. This may be due to test anxiety, demotivation, or that the student has run out of time. A good example can be found in the answers of subject B who "gave up" after the fifth question and provided a correct answer to only the first question.

- **A wrong final answer may be caused by a misconception**

Just as a correct final answer may actually conceal a misconception, a wrong final answer may reveal one or more misconceptions. A misconception may be represented by a mal rule. What distinguishes a misconception is the uniform pattern of problem solving to one or more problems which leads to a wrong answer. Upon a closer inspection of student answers, a list of possible mal rules was found as follows:

mr1	add denominators for the resultant denominator (for common denominator problems)
mr2	add numerators for resultant numerator and added denominators for resultant denominator (for non-common denominator problems)
mr3	add numerators but subtract denominators (for non-common denominator problems)
mr4	add numerators and multiplied denominators (for non-common denominator problems)

Figure 18. A List of Mal Rules

In a perturbation model approach, the mal rules would form part of the bug library and be used to detect misconceptions in student answers.

4.2.1.5 Experiment Summary

This experiment compared two types of diagnosis. A detailed diagnosis of student answers provided more domain-specific information about the student than one which merely characterises each answer as correct or incorrect. A detailed diagnosis revealed the student's problem solving strategies, misconceptions and noisy data such as guesswork, careless slips and copying. This has implications in the way subsequent remedial help is rendered.

4.3 Contents of the Student Model in SKATE

This section discusses the factors which influence the type of information to be maintained in a student model of SKATE. It draws from the findings of the MATT and DSA experiments described in Section 3.2.5 and Section 4.2.1 respectively.

Findings from the MATT experiment show that the expert maintains a detailed model of the student during adaptive testing. Such information included domain-specific information, such as correct and incorrect knowledge, and learner-related characteristics, such as confidence and anxiety. Domain-specific information about the student was specially useful *during* adaptive testing to select subsequent problems and to stop the test, and *after* testing to provide appropriate remedial help. Learner-related characteristics were specially useful to kick-start the test with a problem of appropriate difficulty. Findings from the DSA experiment show that deep cognitive modelling provided more information about the student's cognitive state than a 'shallow' form.

Despite these findings which suggest a detailed student model in terms of domain-specific knowledge, such as misconceptions, and learner-related characteristics, such as confidence, there are other findings from the two experiments and factors which influence the final decision, which is, to include domain-specific knowledge in terms of what he knows. The reasons are discussed as follows.

- **Loose strategy in problem selection**

  In the MATT experiment, it was observed that despite having access to detailed information about the student, the expert did not use the information as effectively as he could have when he employed a rather loose strategy in selecting subsequent questions. Also, there was evidence of redundant questioning. This may mean that the expert did not maintain a detailed model of the students' performance, or it may just have been evidence of human fallibility. It is important to emphasise that this is the strategy of the current expert, and not necessarily one that is used by human testers in general. This also highlights a possible weakness in knowledge acquisition. The human expert may have made use of extraneous data such as signs of student confidence or anxiety. Even if such data was used by the expert human tester, it would not have been possible to exploit it during this project.

- **Access to student's answers to problems**

  A solution to a problem given by a student presents the raw materials for diagnosis. For most adaptive testing systems, it is usual for students to key in only final answers to problems, or to select an answer from a multiple-choice question (Rudner, 1998). The degree of detail that is required from a student in terms of answers to problems depends on the type of diagnosis that is carried out. For example, if the diagnostic element in SKATE requires details on misconceptions, then a student is expected to enter the solution path leading to a final answer. However, as was evident in the DSA experiment, students may provide different degrees of working despite being instructed to display the full working for each question. Therefore a diagnostic module would need to be robust enough to handle not only idiosyncratic answers but should be able to work on different degrees of working given. An alternative is to allow the student to provide a final answer only and to use, like ACM, a machine learning algorithm to work backwards from a final answer to infer a solution path. Limitations of such an approach have been discussed earlier.

- **Task of Constructing a Bug library**

  The ability to perform deep diagnosis usually entails the construction and maintenance of a bug library. As discussed earlier, despite the costly and time-consuming effort in conducting large empirical studies, it is not exempt from the possibility of misdiagnosis, bug migration and the non-generality of bug libraries.

- **Subsequent Remediation**

  In the DSA experiment, it was pointed out that the level of detail of diagnostic information will influence subsequent remediation. However, the remediation strategy of the expert is *reteaching*, as highlighted in Section 3.2.4 and in the MATT experiment (Section 3.2.5). Even in its coarse-grained approach, reteaching is found in some studies to be as effective as remediating errors (Sleeman *et al.*, 1989). Also, Self's slogan of "don't diagnose what you cannot treat" (Self, 1990) can be interpreted to mean that the degree of diagnosis on student answers should be proportional to the degree of subsequent remediation or treatment. This implies that if reteaching is the remediation strategy, then a simple form of diagnosis which characterises student answers as correct or incorrect is sufficient.

From the arguments presented above, the student model in SKATE can be represented as an overlay model and an interaction history component. An overlay model maintains a record of the problem classes which the student has successfully attempted, and an interaction history component keeps a record of the student's successful and unsuccessful attempts to problems.

Like the representation of the problem domain, the overlay student model is *executable*. This is particularly useful for predictive purposes. For example, before a problem is presented to the student, it can be matched against the student model. If a match is found, it can be inferred that the student can solve the new problem. It cannot, however, simulate a student's behaviour at problem solving. An executable model is also useful during remediation where a similar problem can be generated from any problem classes contained within the model and be used as practice exercises for the student.

4.4 The Progression Problem

The issue of progression has been a primary concern in the literature of intelligent tutoring and adaptive testing. In intelligent tutoring, progression involves navigating from one topic to another, or to a problem, or to an exposition, while in adaptive testing, progression is usually in terms of moving from one problem to another, or of meeting the stopping criteria. In both cases, the domain (Halff, 1988) must be structured well enough for the pedagogical module (Anderson, 1988c) or the testing module to make use of. As mentioned previously, a possible drawback is the huge efforts, usually empirical, involved in structuring the domain. For example, bug libraries (Brown and Burton, 1978) and IRT-based test item pools (Wainer, 1990) both require data from large and across different student populations to overcome the problem of non-generality. Also, some domains are constructed with such complexity, for example granularity hierarchies (McCalla *et al.*, 1992), that it may hinder the transfer to other subject domains and the acceptance by end-users such as teachers.

Stochastic methods feature in many problem progression efforts. For example, Marshall (1981), vanderLinden (1998), Villano (1992) and Collins *et al.* (1996) use Bayesian inferencing (Jensen, 1996) (Charniak, 1991) while Huang (1996) employed a probability model based on the IRT. Non-stochastic efforts include the perturbation model approach where problem progression works by first detecting any misconception in a student's answer and matching this misconception against a bug library before presenting a problem to confirm the misconception; an example is the FITS Tutor (Nwana, 1993). Other pieces of information about the student, such as acquisition and retention factors (Beck *et al.*, 1997;Stern *et al.*, 1996), have also been useful in controlling problem progression.

The approach undertaken in this study attempts to avoid some of the major drawbacks of the above techniques, such as large empirical studies and the complexity of domain knowledge construction, by featuring on expert emulation to construct the testing strategy. Two approaches are presented. The first is an algorithm which is designed to investigate the possibility of side-stepping the task of describing the structure of a domain by using expert system knowledge acquisition techniques to elicit from an expert tutor the actual sequence of questions to be used in an adaptive test. A main feature is a query procedure which provides an *authoring* environment for the expert's strategy to be captured. The second explores an elicitation approach based on an analysis of "skills" that a student of a particular problem domain needed in order to successfully solve problems. Although both methods make use of the state-space search paradigm, they are in fact a formalisation of the non-theoretically motivated strategy of a human tutor whose knowledge and skills were worthy of emulation.

4.5 Direct Elicitation of Test Item Sequencing

This section describes a query algorithm for eliciting the test item sequencing strategy of the expert. A manual query approach is tried and this is followed by a computer-aided approach.

4.5.1 Manual Querying An Expert

4.5.1.1 Aim of Experiment

The aim of this experiment is to examine the feasibility of a manual approach in eliciting the test item sequencing strategy of the expert.

4.5.1.2 Method

The expert was given a set of test items and had access to example problems which were generated for each test item. His task was to sequence the set of test items for starting, continuing and stopping the test. In this example, six test items made up the problem domain (Figure 19) which is a smaller set of problem classes than the one presented in Figure 15. Each test item represents a problem class.

a. Add two proper fractions with common denominators
b. Add two improper fractions with common denominators
c. Add a proper fraction and an improper fraction with common denominators
d. Add two proper fractions of different denominators
e. Add two improper fractions of different denominators
f. Add a proper fraction and an improper fraction of different denominators

Figure 19. A Set of Test Items

The interview was conducted along a systematic line of questioning:

> *"Suppose the student has provided a wrong (correct) answer to a problem class X, what would you ask him next?"*

This differs from the style of querying adopted by Kambouri *et al*. (1994) as described in Section 2.5.2.1.

4.5.1.3 Results

The elicited testing sequence takes the form of a binary tree structure (Figure 20). Each node represents a problem class while the arcs dictate the sequence from one node to another, depending on a student's correct or incorrect response.

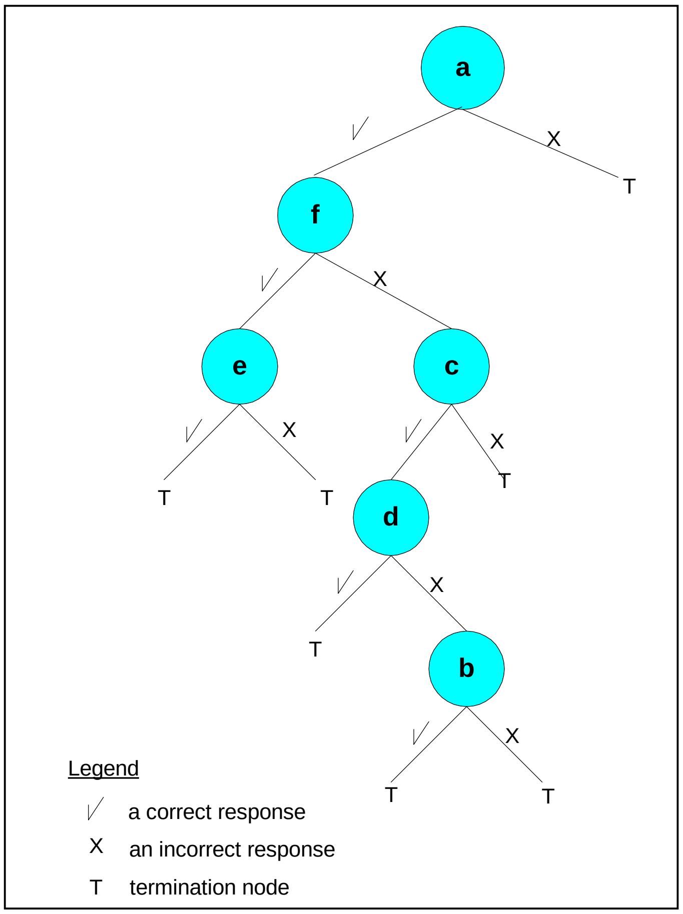

Figure 20. Manually Elicited Test Item Sequence as a Binary Tree

4.5.1.4 Comments

The manual approach was tested for a small set of test items but it is envisaged that this method will break down with increasing number of test items; five to eight test items seem feasible. For example, there is the possibility of noise in terms of contradictions on the part of the expert and the possibility of redundancy in the tree as many paths will share common sequences. An alternative approach is a computer-aided query procedure; this is described in the next section.

4.5.2 Computer-aided Elicitation

A computer-aided approach that facilitates the elicitation of the problem progression strategy of the expert is described. It has two features – a ***query*** procedure which systematically elicits the test item sequence from the expert, and a ***delivery*** procedure which allows the expert to review and confirm his strategy. The query procedure can be used to *author* the adaptive testing component of SKATE where the resultant binary tree called BT can be used by the delivery procedure to function as the adaptive testing strategy of SKATE.

4.5.2.1 The Query Procedure

The query procedure is designed to systematically query the expert in his problem progression strategy and the line of questioning is similar to that of the manual approach described in the previous section. The algorithm for the query procedure is as follows.

Let A be the set of test items {a,b,c,d,e}.

a. When constructing a new binary tree, display A.
b. Expert chooses an item from A and this becomes the entry node for starting the adaptive test.
c. Expert selects a node from which to expand. He selects an item following a correct response, and another following an incorrect response, or a leaf node to indicate a terminating condition.
d. Update branch with new node.
e. Update list of all possible items for expert to choose from.
f. Repeat from c. until no test items are left.

In the example given in Figure 20, each path through the binary tree starts at *a* and ends in a leaf node. Each path corresponds to a possible adaptive test. So, the possible tests are:

a
a,f,e
a,f,c
a,f,c,d
a,f,c,d,b

and the possible outcomes, each is associated with a remediation programme, are:

a, fail
a, succeed, f, succeed, e, succeed
a, succeed, f, succeed, e, fail
a, succeed, f, fail, c, fail
a, succeed, f, fail, c, succeed, d, succeed
a, succeed, f, fail, c, succeed, d, fail, b, fail
a, succeed, f, fail, c, succeed, d, fail, b, succeed

4.5.2.2 The Delivery Procedure – the BT algorithm

The expert can review and confirm the newly elicited problem progression strategy via a delivery procedure. The query and delivery procedures can be run iteratively until the expert is satisfied that the testing sequence is consistent with his strategy. The testing strategy is called BT. The delivery procedure has an additional function of delivering the test during student modelling. The test is administered by traversing the resultant binary tree. The algorithm of the delivery procedure is as follows:

a. make the root of the tree the current node.
b. if the current node is a terminal node, display remedial advice, stop test.
c. ask the question associated with the current node.
d. evaluate the answer provided by the student.
e. if the answer is correct, make the left-hand node the current node.
f. if the answer is incorrect, make the right-hand node the current node.
g. repeat from b.

4.5.2.3 Comments

The computer-aided query procedure may be seen as an improvement over the manual approach in that it can easily keep track of what test items have been used. Such information is useful not only in tailoring the querying process but also in detecting any noise in terms of contradictions from the expert. However, the number of possible questions increases with the number of test items and this may increase the possibility of contradictory behaviour on the part of the expert. This was a problem faced also by Kambouri *et al.* (1994), as described in Section 2.5.2.1, where experts judged between 1000 and 2500 questions for a 50-item problem domain and it was already considered an improvement over the $2.8\text{x}10^{16}$ possible questions.

A student model can be maintained to keep diagnostic information about the student and facilitate subsequent remediation. The delivery procedure can be augmented by a diagnostic module to record the path taken by a student through the binary tree of problem classes. For example, a path can be one of the possible outcomes described earlier and each leaf node of the binary tree is a remedial programme. The binary tree is a means of mapping each student onto a programme.

4.6 Problem Progression based on Problem Solving Skills

This section describes the elicitation of the test item sequence of the expert based on the expert's measure of problem difficulty (see Section 3.7.2). The strategy called XP is an alternative to the one proposed in the previous section.

4.6.1 Aim of Experiment

The aim of the experiment is to elicit a strategy in problem progression based on the expert's assertion that the difficulty of a problem can be measured by the number of problem solving skills needed to solve the problem. The expert has taken the assumption that all the skills under consideration are of the same level of difficulty.

4.6.2 Method

Knowledge elicitation took the form of interviewing and task analysis and was carried out using the domain of fraction additions with five problem solving skills (see Section 3.7.1). The general approach derived from the interviews with the expert is as follows. Sets of problems requiring a specific number of skills to be used by the student are formed. The sets are labelled with a number. For example, in Figure 21, the node with a number, say 3, denotes a set of problems which each requires specifically three skills to be solved. A process rather like a binary search is used to investigate a student's ability. Having established either competence or incompetence at set *n*, the next set of problems to be investigated are set mid way between *n* and *max* (for competence) and *n* and *min* (for incompetence) where *max* and *min* are the largest and smallest labels. The process is repeated with revised values for *max* and *min* as appropriate. The exploration of problems in a particular set is explained by the example in the next section, with *max* equals 5 skills. The strategy of the expert was also captured for different values of *max*.

4.6.3 Example

The example was extracted from a knowledge elicitation session. In Figure 21, the adaptive test begins at node 3 that contains problems each of which can be solved by three skills.

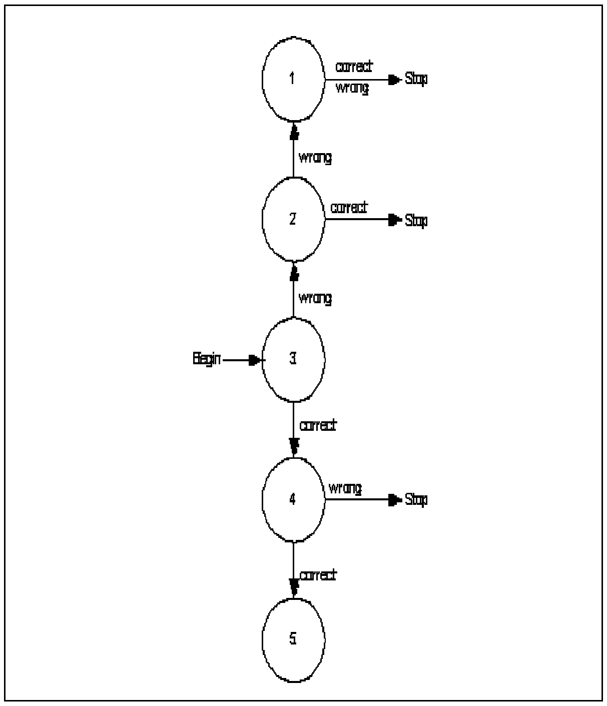

Figure 21. Problem Progression for a Domain of Five Skills

Problem progression from node to node, that is from one level of difficulty to another, works like this. If the student gets more incorrect than correct answers to problems within that category, he moves onto node 2 which contains problems each of which can be solved by exactly two skills. Conversely, if he gets more correct answers within node 3, he will move onto node 4 which contains problems each of which can be solved by exactly four skills.

Problem progression *within* a node works like this. If there are a set of five skills, {a,b,c,d,e}, then at node 3, say, there are ${}^{n}C_{r}$ possible combinations of skills, that is:

$${}^{n}C_{r} = \frac{n!}{(n-r)!r!}$$

or ${}^{5}C_{3}$ or 10 possible combinations of skills: {[a,b,c], [a,b,d], [a,b,e], [a,c,d], [a,c,e], [a,d,e], [b,c,d], [b,c,e], [b,d,e], [c,d,e]}. For example, the combination [a,b,c] would involve a set of problems which each require all the skills *a*, *b* and *c* to be used. Some combinations may not yield a problem. For example, there may be no problems associated with combination [a,d,e].

As a skill can appear more than once in different combinations, the expert decided to give priority to those skills which have been asked the least. This criteria was enforced by the knowledge engineer through the use of weights where weights were introduced to each combination to enable the selection of the next best combination. The following criteria were imposed for calculating the weight of each candidate set:

- If a skill has been not been asked yet, it carries a weight of 2
- If a skill has already been asked once, it carries a weight of 1
- If a skill has been asked more than once, it carries no weight

Based on the weighting criteria, the following example shows how problems, each of which requiring a combination of three skills, were presented to the student.

a. Select the first set amongst the list of candidate combinations, in this case, [a,b,c]. Calculate the scores of the other combinations, based on the above rules.

	[a,b,c	[a,b,d]	[a,b,e]	[a,c,d]	[a,c,e]	[a,d,e]	[b,c,d]	[b,c,e]	[b,d,e]	[c,d,e]
1	*	4	4	4	4	5	4	4	5	5

b. Select the first set with the highest score. Combination [a,d,e] is chosen and the scores of the other combinations are recalculated.

	[a,b,c]	[a,b,d]	[a,b,e]	[a,c,d]	[a,c,e]	[a,d,e]	[b,c,d]	[b,c,e]	[b,d,e]	[c,d,e]
1	*	4	4	4	4	5	4	4	5	5
2	-	2	2	2	2	*	3	3	3	3

c. Combination [b,c,d] becomes the next best choice and is thus chosen.

	[a,b,c]	[a,b,d]	[a,b,e]	[a,c,d]	[a,c,e]	[a,d,e]	[b,c,d]	[b,c,e]	[b,d,e]	[c,d,e]
1	*	4	4	4	4	5	4	4	5	5
2	-	2	2	2	2	*	3	3	3	3
3	-	0	1	0	1	-	*	1	1	1

d. Combination [a,b,e] becomes the next best choice and is chosen.

	[a,b,c]	[a,b,d]	[a,b,e]	[a,c,d]	[a,c,e]	[a,d,e]	[b,c,d]	[b,c,e]	[b,d,e]	[c,d,e]
1	*	4	4	4	4	5	4	4	5	5
2	-	2	2	2	2	*	3	3	3	3
3	-	0	1	0	1	-	*	1	1	1
4	-	0	*	0	0	0	0	0	0	0

e. As there are no more candidate sets, no more problems are presented.

In the above example, it shows that out of the ten possible combinations, only problems of combinations [a,b,c], [a,d,e], [b,c,d] and [a,b,e] were chosen.

Progression to the next node depends on the student's performance at the current node. If the student had provided any incorrect responses, he would be assigned 'easier' problems; in this case, this means progression to node 2, where he would be presented with problems requiring exactly two skills. The whole process is repeated for problems requiring exactly two skills. Conversely, if the student had answered all the problems at the current node 3 correctly, then he would assign 'harder' problems (node 4) each of which requires exactly four skills.

It must be noted that it is possible that not every combination would yield a problem.

4.6.4 Comments

- **Pruning the search space within a node**

Pruning the search space within a node takes place in two ways. Firstly, not all the generated combinations will yield a valid problem. Secondly, the weighting criteria are used to constrain the choice of future problems. Constraining future choices in this way is similar to the inferences used by Kambouri *et al.* (1994) to reduce the number of knowledge states, or the technique described in the previous section to reduce the amount of effort needed to construct a binary tree of test. In this way, the intuitively attractive idea of not repeatedly gathering information about the same skills can be operationalised.

- **Pruning the search space from one node to another**

The expert took the view that if a student has solved a 'harder' problem, it can be implied that he could solve an 'easier' problem. In this way, the set of candidate problems could be pruned further. For example, if a student has successfully solved problems requiring three skills say, then he need not be presented with problems requiring a lesser number of similar skills. This conforms with the notion of "downward consistency" (Section 4.2.1.4).

- **The Use of a Student Model**

Throughout the adaptive testing, a record of the student's performance is maintained as an interaction history component of the student model. The student model is constantly being updated throughout the adaptive testing to record the problems which the student has already tackled, and this information was used to guide the selection of future problems and subsequent remediation.

4.7 Conclusion

The chapter discussed the experiments leading to the creation of a student model and a problem progression strategy for SKATE. The findings of two experiments led to the choice to maintain domain-specific information about the student in terms of an overlay model and interaction history module. The clp(FD) representation of the overlay model meant that it is executable and is useful for predicting a student's performance and for generating problems to aid remediation. In problem progression, two options for the design of a module to control the sequence of test items in an adaptive test were presented. The first experiment demonstrated the scope for computer assistance in knowledge elicitation while the second experiment adopted a skills-based approach based on the expert's measure of problem difficulty. The two strategies made use of a student model for diagnosis and remediation and in selecting an approach suitable for SKATE, testing systems associated with student modelling are preferred to those developed for summative testing purposes only, such as IRT-based systems.

Chapter 5.

Design and Implementation

5.1 Introduction

The elicitation carried out in the last three chapters resulted in the construction of the three main knowledge components of SKATE, namely the domain knowledge, the student model and the adaptive testing strategy. Two testing strategies, XP and BT, were elicited from the human expert.

This chapter discusses the contents of the student model and the domain knowledge in relation to XP (see Section 4.6), which was the preferred testing strategy of the domain expert. Fraction addition was the example domain used throughout the study but the proposed architecture should be able to support other topics in mathematics. The design of each component is the result of the aim to produce adaptive tests that are both efficient and acceptable to teaching professionals. Each component has been strongly influenced by interaction with a domain expert. This is described in a section on “Origins of the Design” which follows a general overview of the structure of the model. Subsequent sections are devoted to describing each of the main components of the model.

5.2 Origins of the Design

The process of Knowledge Acquisition has been described in Chapter 3. Since the domain expert is the authority for both the domain knowledge and the testing strategy, he is the major influence on the student modelling architecture. The approach based on emulation to adaptive testing was, by design, entirely under the control of the expert. Although the subject matter of the research, “fractional arithmetic”, is well understood, this does not mean that there is only one way in which domain knowledge can be represented. The way in which this emerged, with an emphasis on specific manipulative skills, and a concern that skill combinations should be the only measure of problem difficulty and that problem difficulty should be central to testing, again reflected the influence of the domain expert. This means that there is no reason to suppose that an identical testing strategy and domain modelling would emerge were the work to be repeated with a different expert. But some elements of the model would be constant, and more significantly, the approach to constructing the model would remain the same.

The domain expert’s characterisation of fractional arithmetic in terms of specific manipulative skills leads directly to the design of the student model. It is the ability to exercise skills in problems at a certain level of difficulty that represents a student’s competence. The student model in SKATE, for this particular domain, consists, at any time during the testing process, of those skill combinations that the testing strategy indicates are within the ability range of the student. Implicit in the expert’s approach to testing is an assumption of a hierarchy of competence. This means that if a student can exercise the set of skills {a,b,c} to solve a problem, then it follows that the student can exercise any subset of those skills, so that there is no need to explicitly test for competence with those subsets. However, competence with {a,b,c} has no implication for competence with proper super sets. It is this hierarchical assumption that motivated the expert’s testing strategy, and a way of summarising the ability of a student in terms of the maximum size of the set of skills in which he had demonstrated competence.

5.3 SKATE – A Student Modelling Architecture

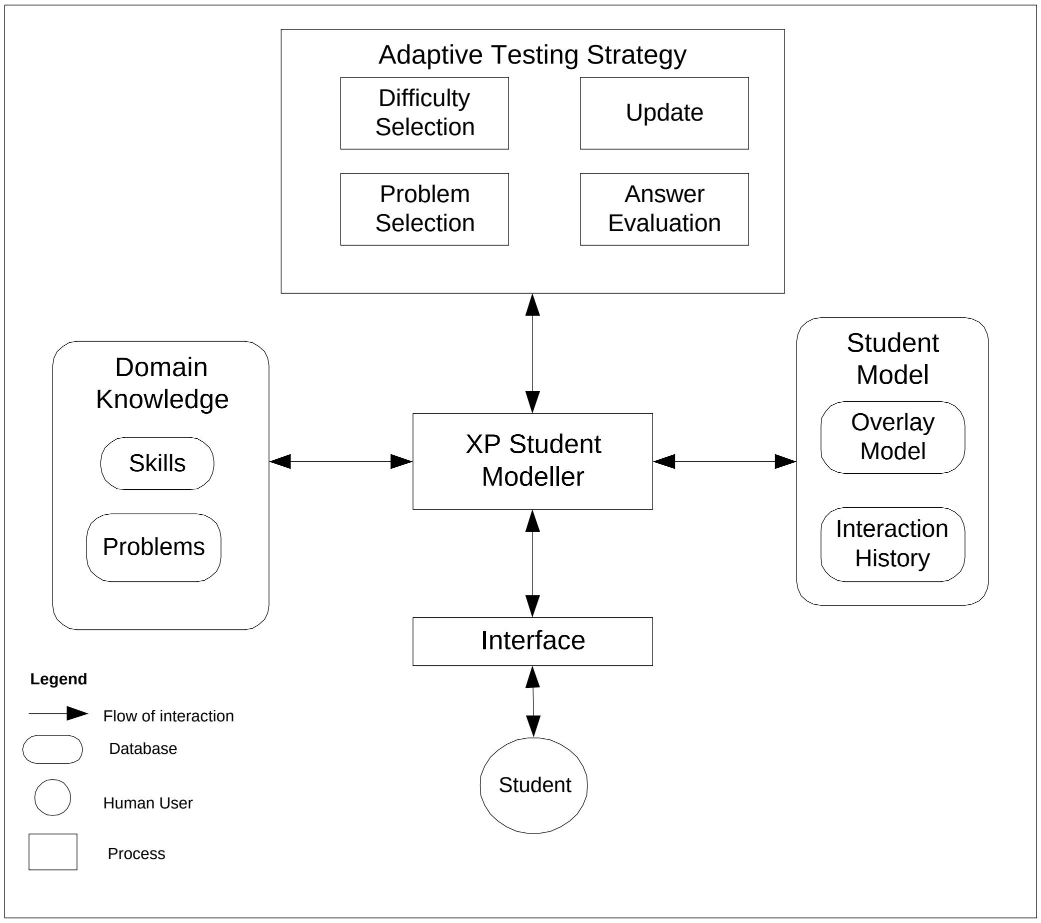

Figure 22. The Architecture of SKATE with XP testing strategy

The architecture of SKATE was first introduced in Chapter 1. A more detailed architecture is given in Figure 22, with XP as the kernel of the SKATE model. The *student modeller* orchestrates the student modeling process. First, the *difficulty selection* module is invoked to determine the level of difficulty of the first problem to be presented. With the assigned difficulty level, the *problem selection* module selects a problem from the *problems* bank in the *domain knowledge* component. The problem is presented to the student via a simple *interface*. The student’s response is passed to the *answer evaluation* module, which compares the student’s response with that of the system. The verdict is recorded in the *interaction history* module by the *update* module. A predetermined number of problems within the assigned difficulty level are presented to the student, one at a time, with no duplicates of problems or skill combinations. Once this is exhausted, the cycle repeats with subsequent selection of difficulty level being determined by how the student fared in answering the set of questions at the current difficulty level. The test stops at a difficulty level which indicates the student's level of attainment.

5.4 The Adaptive Testing Strategy

The main functions of XP are described in more detail.

- **Difficulty Selection**

  This is a binary chop algorithm to determine progression from one level of difficulty to the next. Problem difficulty is proportional to the number of skills needed to solve the problem. All skills are considered to be equally difficult (see Section 3.7.2). The rule used to determine movement from one level to the next is strict. The student has to demonstrate competence by solving all the problems presented at the assigned difficulty level. He is only allowed to progress to a higher difficulty level if he has provided more correct answers than wrong ones at the present level. The number of questions at each level was determined by the domain expert during the Knowledge Acquisition process. Conversely, if he has more wrong answers than right ones, a lower difficulty level is assigned where he will be presented with problems requiring fewer skills to be applied.

- **Problem Selection**

  The selection is based on matching a combination of skills to a problem. This approach requires the presentation of problems of varying number and combination of skills. As an example, say there are five problem solving skills for a domain under study, [a,b,c,d,e], and the present difficulty level is 2, then all two skill problems in the question bank become candidates for selection. No problem is presented more than once. Also, any problem which requires the same combination of skills, that is, the same skills in the same order to be exercised, is removed. For example, if there are three problems, *q1*, *q2* and *q3*, which require the same order of skills to be applied, say [b,c], then *q2* and *q3* no longer become candidates for selection.

- **Answer Evaluation**

  The answer from the student, in response to a problem, is checked against that of the system. The system's answers to problems are recorded in the *problems* bank. If a match is found, it returns the verdict as 'correct', otherwise the verdict 'wrong' is returned.

- **Update**

  The *update* module records in the interaction history module, details of each student's attempts on problems presented to him. The details include the student's answer, the system's verdict and the combination of skills which the system believes the student has exercised. At the end of the test, the *update* module summarises which skills the student is believed to have mastered and this is recorded in the *overlay* student model.

5.4.1 Parameters of XP

The parameters under which XP operates are described.

a. Number of problems per combination of skills

When XP determines a set of skills out of ${}^{n}C_{r}$ possible combinations of skills (see Section 4.6.3), a problem matching that combination is chosen from the problems database and presented to the student. There is usually at least one problem for any combination of skills. XP currently requires the presentation of only one problem per combination of skills.

b. Progression of difficulty level

Each student starts the test at the same level of difficulty. With the version of the binary chop algorithm currently adopted by XP, as discussed in Section 4.6, progression from one level of difficulty to the next is calculated by *rounding up* to the nearest integer, the value of the midpoint between the present level of difficulty and the highest or lowest level of difficulty, depending on whether progression is to easier or more difficult problems. For example, in a six-skill problem domain, the starting level of difficulty is 4, which is the rounded up value of the midpoint between 1 and 6. If the student proceeds to easier problems, the next difficulty level will be 3.

c. Threshold level of success

A student progresses to a higher (lower) difficulty level when he provides more correct (incorrect) answers to problems than incorrect (correct) ones at the present level of difficulty. The difference between the number of correct and wrong answers may be no greater than 1.

d. No intervention

XP performs student modelling via problem presentation. Unlike the more generic types of Intelligent Tutoring Systems, it does not intervene to offer hints or explanation.

e. Low Bandwidth

Like PROUST (Soloway and Johnson, 1984), XP has access to student's final answers only, and not intermediate steps or solution paths. This is different from performance assessment that includes data on student problem solving behaviour as well as their answers, as described by Linn, Baker and Bunbar (1991) in VanLehn & Niu (2001). With such low bandwidth of information, XP performs knowledge tracing and not model tracing.

f. No Memory of previously answered questions

XP has no memory of previously answered questions. It does not have the mechanism of reviewing previously answered questions to check for any inconsistencies, which could arise from noisy data such as careless slips or lucky guesses. This is further discussed in Section 6.6.

g. One solution path per problem

There may be more than one solution path leading to the correct final answer of a problem. XP currently stores one solution path per problem. A solution path is represented as an ordered combination of skills which can be exercised to arrive at the correct final answer.

Different versions of XP can be created by altering one or more of the above parameters. This is taken up in Sections 6.7 and 6.8.

5.5 Domain Knowledge

Domain knowledge in SKATE is concerned with problem solving skills and problems. It is the problem solving skills as exhibited by students that are the object of the adaptive tests. The next subsections describe the problem solving skills relevant to the subject domain under study and a typography of problems derived from the exercise of skills.

5.5.1 Problem Solving Skills

The problem solving skills identified during knowledge acquisition and discussed in Section 3.7.1 can be further reclassified as the following:

- ***makeVulgar***
  This skill transforms a mixed fraction into an improper fraction, or returns proper and improper fractions unchanged.
- ***makeCommon***
  This skill manipulates fractions to give a common denominator.
- ***checkAndAdd***
  This skill adds two proper fractions of a common denominator.
- ***cancel***
  This skill removes common factors from the numerator and denominator of a fraction and the resultant is a fraction is in its lowest form.
- ***makeProper***
  This skill transforms an improper fraction into a mixed fraction.
- ***makeWhole***
  This skill provides a whole number, 1 or 0, by applying the *makeProper* skill to fractions such as 3/3 and 0/3.

The reclassification allows the introduction of a three-phase structure, ***Prepare-Add-Tidy***, which are the three phases involved in solving fraction addition problems. The set of problem solving skills is represented in SKATE as the predicate, *skills(N, L)*, where *N* is the total number of skills identified for the topic under study and *L* is the list of identifiable skills. An example for the current subject domain is:

skills(6, [*makeVulgar, makeCommon, checkAndAdd, cancel, makeProper, makeWhole*]).

Each problem solving skill can be categorised into the respective phase as follows:

Prepare	**Add**	**Tidy**
makeVulgar *makeCommon*	*checkAndAdd*	*cancel* *makeProper* *makeWhole*

For example, to solve the problem 1/12 + 1/6 requires the use of the *makeCommon* skill in the *Prepare* phase to transform the task to 1/12 + 2/12. The *checkAndAdd* skill in the *Add* phase can then be exercised to yield 3/12. Using the *cancel* skill in the *Tidy* phase, this can be transformed to 1/4. Each phase can contain one or more skills. It is assumed that a skill is used in not more than one phase.

It will be shown in Section 6.3 how the three-phase structure is used to describe the overlay or buggy student model of a simulated student.

5.5.2 Problems

Having represented a set of problem solving skills, the XP algorithm requires problems to be characterised by the opportunity they provide for the exercise of the skills. Problems can be either predicates that generate problems, similar to the approach of the BT algorithm (Section 4.5.2), or can be hand-coded by a human assessor, as was the case with the DSA experiment (Section 4.2.1). Each problem is passed through a rule-based problem solver to produce the final correct answer and the corresponding set of skills needed to solve the problem. A sample of problems is given in Appendix H where each problem is represented by the predicate:

$$question(N, C_p, T, P, A_p)$$

where *N* is the difficulty level representing *N* number of skills needed to solve question *P* of type *T*, by using a combination of skills, C_p. The combination C_p is the set of skills provided by the problem solver. The system's correct answer is A_p. Examples are:

question(2,[c,e], q_ce_6, fr(7/5,2/5), fr(1:4/5)).
question(3,[a,c,e], q_ace_1, fr(1:1/5,2/5), fr(1:3/5)).

Problem progression within a difficulty level was described in Section 4.6.3. There may be no problem associated with certain ${}^{n}C_{r}$ combinations of skills. For example, there is no fraction addition problems associated with the six-skill combination [*makeVulgar, makeCommon, checkAndAdd, cancel, makeProper, makeWhole*]. If this occurs, SKATE will proceed to another combination.

The following subsections address the generation of different problems, characterised by the number of skills required to solve them.

5.5.2.1 One Skill Problems

There are some problems that can be completely solved with one skill, namely *checkAndAdd* in the *Add* phase. Examples are 1/3 + 1/3 = 2/3 and 3/5 + 1/5 = 4/5. The characteristics of such problems are common denominators, the sum of the numerators is less than the denominator and the sum of numerators and denominators do not have common factors, that is, they are prime numbers.

5.5.2.2 Two Skills Problems

Examples of valid two skill combinations are [*checkAndAdd, cancel*], [*checkAndAdd, makeProper*], and [*makeCommon, checkAndAdd*]. If we allow the *cancel* skill to be used outside the *Tidy* phase, we could allow 2/6 + 1/3 which can be transformed, using the *cancel* skill, into 1/3 + 1/3, before applying *checkAndAdd* to arrive at 2/3.

Examples of two skill combination problems using *checkAndAdd* and *cancel* are:

1/6 + 1/6 = 2/6 = 1/3
3/8 + 1/8 = 4/8 = 1/2

The characteristics of such problems are common denominators, the sum of numerators is less than the denominator and the sum of numerators and denominators has a common factor.

Examples of two skill combination problems using *checkAndAdd* and *makeProper* are:

2/3 + 2/3 = 4/3 = 1:1/3
5/6 + 2/6 = 7/6 = 1:1/6
7/5 +2/5 = 9/5 = 1:4/5

The characteristics are common denominators, the sum of numerators is less than the denominator and the sum of numerators and denominators does not have a common factor.

Examples of two skill combination problems using *makeCommon* and *checkAndAdd* are:

1/3 + 1/5 = 5/15 + 3/15 = 8/15
5/8 + 1/6 = 15/24 + 4/24 = 19/24

The characteristics of such problems are no common denominator, the sum of numerators is less than the denominator, and the sum of numerators modulo the denominator and the denominators does not have a common factor.

5.5.2.3 Three Skills Problems

Valid three skill problems are [*makeCommon, checkAndAdd, Cancel*], [*makeCommon, checkAndAdd, makeProper*], [*checkAndAdd, makeProper, makeWhole*] and [*makeVulgar, checkAndAdd, makeProper*].

Examples of problems using skills *makeCommon*, *checkAndAdd* and *cancel* are:

1/12 + 1/6 = 1/12 + 2/12 = 3/12 = 1/4
2/15 + 1/5 = 2/15 + 3/15 = 5/15 = 1/3
3/15 + 2/5 = 3/15 + 6/15 = 9/15 = 2/3

The characteristics of such problems are no common denominator and the sum of fractions is always less than 1.

Examples of problems using skills *makeCommon*, *checkAndAdd* and *makeProper* are:

4/5 + 3/4 = 16/20 + 15/20 = 31/20 = 1:11/20

5/6 + 1/3 = 5/6 + 2/6 = 7/6 = 1:1/6

6/7 + 3/8 = 48/56 + 21/56 = 69/56 = 1:13/56

6/7 + 5/8 = 48/56 + 35/56 = 83/56 = 1:27/56

5/8 + 4/5 = 25/40 + 32/40 = 57/40 = 1:17/40

5/8 + 5/6 = 15/24 + 20/24 = 35/24 = 1:11/24

3/8 + 5/6 = 9/24 + 20/24 = 29/24 = 1:5/24

8/9 + 3/5 = 40/45 + 27/45 = 67/45 = 1:22/45

The characteristics of these problems are no common denominator, the sum is always greater than 1, and the numerator and denominator of the factional part of sum have no common factor.

Examples of problems using skills *checkAndAdd*, *makeWhole*, *makeProper* are:

4/5 + 1/5 = 5/5 = 1:0/5 = 1

1/2 + 1/2 = 2/2 = 1:0/2 = 1

Characteristics of such problems are common denominators and the sum is 1.

Examples of problems using skills *makeVulgar*, *checkAndAdd*, *makeProper* are:

1:1/5 + 2/5 = 6/5 + 2/5 = 8/5 = 1:3/5

2: 3/7 + 2/7 = 17/7 + 2/7 = 19/7 = 2:5/7

Characteristics of these problems are common denominators, at least one operand is greater than 1 and the numerator and the denominator of the factional part of sum have no common factor.

5.5.2.4 Four Skills Problems

Valid four skill combinations are [*makeVulgar, checkAndAdd, makeProper, makeWhole*], [*makeCommon,checkAndAdd,makeProper,cancel*],[make*Vulgar, checkAndAdd, makeProper, cancel*], [*makeVulgar, makeCommon, checkAndAdd, makeProper*].

Examples of problems using skills *makeVulgar*, *checkAndAdd*, *makeProper*, *makeWhole* are:

1:1/3 + 2/3 = 4/3 + 2/3 = 6/3 = 2: 0/3 = 2

1:1/5 + 3/5 = 6/5 + 4/5 = 10/5 = 2:0/5 = 2

Characteristics of these problems are at least one operand is greater than 1, common denominators and the numerator and the denominator of the factional part of sum must be a multiple of the denominator.

Examples of problems using skills *makeCommon*, *checkAndAdd*, *cancel* and *makeProper* are:

3/8 + 5/6 = 18/48 + 40/48= 58/48 = 29/24 = 1:5/24

4/5 + 3/10 = 40/50 + 15/50 = 55/50 = 11/10 = 1:1/10

7/8 + 3/4 = 28/32 + 24/32 = 52/32 = 13/8 = 1:5/8

Characteristics of such problems include operands less than 1, sum is always greater than 1 and the common denominator is greater than the lowest common denominator.

Examples of problems using skills *makeVulgar*, *checkAndAdd*, *cancel* and *makeProper* are:

1:1/8 + 1:3/8 = 9/8 + 11/8 = 20/8 = 5/2 = 2:1/2

1:1/6 + 2:1/6 = 7/6 + 13/6 = 20/6 = 10/3 = 3:1/3

Characteristics of such problems are at least one operand is greater than 1, common denominators and the numerator and the denominator of the factional part of sum must have a common factor.

Examples of problems using skills *makeVulgar*, *makeCommon*, *checkAndAdd*, *makeProper* are:

> 1:3/8 + 1/2 = 11/8 + 1/2 = 11/8 + 4/8 = 15/8 = 1:7/8
> 1/5 + 1:5/7 = 1/5 + 12/7 = 7/35 + 60/35 = 67/35 = 1:32/35

Characteristics of these problems are at least one operand is greater than 1 and the denominators are relatively prime.

5.5.2.5 Five Skills Problems

Valid five skill problems are [*makeVulgar,makeCommon,checkAndAdd,cancel makeProper,*]. An example is:

> 1/3 + 1:5/6 = 1/3 + 11/6 = 6/18 + 33/18 = 39/18 = 13/6 = 2:1/6

Characteristics of such problems are that an operand is greater than 1, the denominators are not the same and the numerator and the denominator of the factional part of sum must have a common factor.

5.6 The Student Model

The student model contains the system's beliefs of the student's level of knowledge of the subject domain. These are inferred from the interaction with the student during the test and are made up of two types of domain-specific information:

- an ***overlay model*** which is a set of problem solving skills that the system believes the student has mastered. It can be a subset or the whole set of the problem solving skills identified in the subject domain.

- an ***interaction history*** which is the set of problems presented to the student together with the system's verdicts of the student's answers, the difficulty level and the combination of skills provided by the problem solver. This is represented as the predicate, *visited*(N,C_p,T,P,V), where N is the difficulty level, C_p the combination of skills which the system believes the student has exercised to solve problem P of type T. The parameter C_p is the same as C_p of the *questions/5* predicate described in Section 5.5.2. The system's evaluation of the student's answer is represented as *V*. An example is given below.

```
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
```

Figure 23. A Fragment of the Interaction History Module

The above fragment shows successfully and unsuccessfully exercised sets of skills at different levels of difficulty. During student modelling, the *visited/5* predicate is checked for two purposes. Firstly, it is to ensure that the same problem or combination is not presented more than once. Secondly, it is used to determine progression to the next level. In the above illustration, as the student has produced more incorrect than correct answers at level 3, he proceeds to a lower level of difficulty, that is, 2.

Other types of information can be derived from the *visited/5* predicate, as will be discussed in the next chapter.

5.7 Conclusion

This chapter described the student modelling architecture, SKATE. It is based on the information acquired from an expert teacher of elementary arithmetic. The underlying domain knowledge relates to a particular view of the topic by a particular person. The approach to adaptability developed in SKATE is not the only technique used by the domain expert. It has the advantage that it can be used as a basis for a series of experiments that charts the consequence of a range of minor variations. An account is given of such a range of experiments using a simulation of student performance to follow through the consequences of such changes in the next chapter.

Chapter 6.

Experiment and Analysis

6.1 Introduction

Effective evaluation of educational software requires trials with students. A common but time-consuming approach is to have a human assessor analysing verbal and written protocols of human students who were solving a large number of problems. An alternative way is to perform evaluation using simulation. It is a convenient way of exploring the workings of an adaptive testing strategy where it is possible to make comparisons at various levels of student performance, between students at various predetermined ability levels.

This chapter presents an evaluation of the XP testing strategy or assessor, as it shall be called in this chapter, through the use of *simulated students*. A three-phase structure, *Prepare-Add-Tidy,* designed in the previous chapter, is used to describe a simulated student. An added feature is the introduction of malrules into the three-phase structure. The results from the evaluation of XP are analysed and compared with those of a sequential file, called ST. By varying the parameters of XP, the performance of XP was re-evaluated and compared with ST.

The chapter is structured in the following way. The evaluation strategy is described. This is followed by a description on how evaluation is being carried out and a discussion on experimental results. The varying parameters of XP are described followed by an evaluation of the performance of the different versions of XP.

6.2 The Evaluation Strategy

A method for evaluating a student modelling strategy is to develop a model of a student, use it to predict its performance and to check to see if the prediction is accurate. This was the method undertaken by Anderson et al. (1995) in the evaluation of the LISP Tutor and by Shute (1995) in evaluating the SMART student modelling system. More recently, VanLehn & Niu (2001) presented an evaluation of the ANDES assessor through the use of a set of simulated students which was generated to depict varying conditions in which to test the performance of the assessor.

A *simulated student* is a computer model of a human student (VanLehn, Ohlsson & Nason 1994). It assumes the traits and characteristics of a human student, which are in this study, confined to overlay or buggy knowledge with no conative or affective characteristics. VanLehn, Ohlsson & Nason (1994) identified three main applications of student simulation. There are tutor training systems, collaborative learning and formative evaluations. An example is a three-agent learning situation (Chan & Baskin 1990) where interaction takes place between a computer tutor, a human learner and a simulated learning companion. The simulated learning companion can also take the form of a ‘troublemaker’.

Like the work of VanLehn & Niu (2001), simulated students are used in this study in place of human students mainly for the ease of performing an evaluation on a student modelling system. In their work, a set of solution graphs of Physics problems is converted into Bayesian Belief Networks (or BBN). A problem solver of ANDES generates the solution graphs. A simulated student is generated by randomly deleting rules from the BBN, thus modelling the fact that different students have different knowledge. The reduced BBN is then used to solve problems. The problem solving actions are recorded and passed to the ANDES assessor, which predicts the mastery level of each rule. The assessor is deemed accurate if it assigned the deleted rules a low posterior probability of mastery and the other rules a high probability.

There are many advantages of using simulated students. Firstly, a simulated student can be readily created by assuming as many or as few traits, which means that its competence is known. This makes it easy to determine the accuracy of the assessor in its prediction of the simulated student’s competence. Secondly, an ideal setting for any computerised adaptive testing is one in which its students do not suffer from anxiety which could distort the accuracy of the test. This setting could be achieved more easily through the use of simulated students than a test setting of real students.

Evaluation is carried out in the following way. Different types of students are simulated. A set of problems, in the domain of fraction additions, is prepared and solved by each simulated student, creating a series of log files. Each log file is assessed in a series of runs by an assessor which infers the mastery of skills of each student. The following steps in the evaluation are identified as:

1. Creating simulated students
2. Generating Logfiles
3. Running the assessor

Figure 24 shows the process of evaluation. A set of simulated students called *Sam* is created, each equipped with a problem solver and an overlay or buggy student model. Each student is required to solve all the questions in the problems bank and their attempts are recorded in logfiles. The logfiles are then used by the assessors to infer an overlay student model and to create a student interaction history for each student. It is useful to note that the assessors are run in ‘batch’ mode, and not interactively with each simulated student. Each inferred overlay student model of ST and XP is matched against the overlay model, and not buggy knowledge, of the corresponding simulated student. The accuracy of the assessor is rated by the number of matches that can be found. A more precise measurement of the performance of the assessor is conducted by analysing each student interaction history.

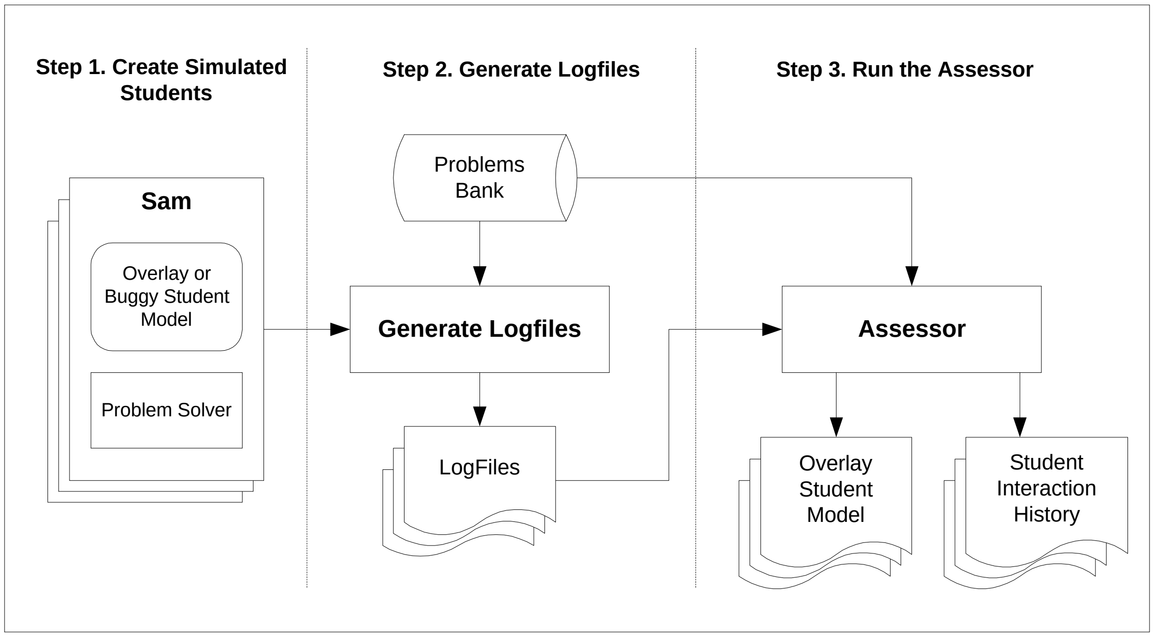
Step 1. Create Simulated Students
Step 2. Generate Logfiles
Step 3. Run the Assessor
Sam
Overlay or Buggy Student Model
Problem Solver
Problems Bank
Generate Logfiles
LogFiles
Assessor
Overlay Student Model
Student Interaction History

Figure 24. The Evaluation Strategy

The following sections describe the evaluation steps in more detail.

6.3 Creating Simulated Students

As mentioned earlier, a simulated student consists of two parts - a problem solver and an overlay or buggy student model. The task of the problem solver is to solve problems using the overlay or buggy student model. The student model is built on a three-phase, *Prepare-Add-Tidy,* structure. This format, introduced in Section 5.5.1, allows different types of students to be simulated. In this study, five types of students are simulated. They are *Sam1, Sam2, Sam3, Sam4 and Sam5*. They will be described in the following subsections.

Different simulated students can be created by instantiating from any type of student listed above. In this study, 28 such students are simulated and their student models represented as Prolog predicates, as shown in Appendix G. The format is:

simStudents(S, [S-L_s])

where *S* is the name of the simulated student and L_s is a list of mastered skills and/or malrules. L_s is made up of three parts corresponding to the three-phase structure of a simulated student.

The assumption is that no forgetting or learning takes place during assessment. This is similar to ANDES (VanLehn and Niu 2001). For example, if a student knows all the relevant skills, he will exercise them as required during problem solving.

6.3.1 Sam1 Student Type – knows all the skills

This type of student knows all the relevant skills in the subject domain and can correctly apply them in order to solve a problem. He may not need to exercise all the skills at one time to solve a problem. Also, apart from mastering all the skills, he must be able to execute them in an appropriate order for successful operation. For example, if *sam1a* and *sam1c* are two simulated students who know all the relevant skills but in a different order from one another:

```
simStudents(sam1a,[[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
simStudents(sam1c,[[sam1c-[[makeVulgar,makeCommon],[checkAndAdd],[makeProper,makeWhole,cancel]]]]).
```

This may have an effect on problem solving. For example, *sam1a* may be able to produce the correct answer for a question while *sam1c* may not.

6.3.2 Sam2 Student Type – gaps in knowledge

This type of student has gaps in his knowledge of the subject domain. This means that there are one or more skills he does not know or has not yet mastered. He solves problems using the skills that he knows. An example is *sam2a* who knows all the skills except the *makeCommon* skill:

```
simStudents(sam2a,[[sam2a-[[makeVulgar],[checkAndAdd],[cancel,makeProper,makeWhole]]]]).
```

6.3.3 Sam3 Student Type - malrules

This type of student has some buggy knowledge or malrules. Malrules or misconceptions are rules, perhaps invented by the pupil, which appear effective but in fact work only under certain conditions (Hall 2002). An example is *sam3a* who has mislearned the *cancel* skill and stored this as a malrule called *malCancel*:

```
simStudents(sam3a,[[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel, makeProper, makeWhole]]]]).
```

6.3.4 Sam4 Student Type – lucky guesses

This type of student can make lucky guesses which to the assessor, are synonymous to copying. He may have gaps in his knowledge, like *Sam2*, or malrules, like *Sam3*, but somehow manages to produce correct final answers to questions which require the application of certain skills that he has not mastered or has misconceptions on. An example of *Sam4* type of student is *sam4a:*

```
simStudents(sam4a, [[sam4a-[[makeCommon],[checkAndAdd],[]]]]).
```

Student *sam4a* has an identical overlay model to *sam2f*. In this study, the logfiles of *sam4a, sam4b* and *sam4c* were created by 'tweaking' the logfile of *sam2f* to simulate lucky guesses. The logfiles of *sam4d* and *sam4e* were created by 'tweaking' the logfiles of *sam2a* and *sam2c* respectively.

6.3.5 Sam5 Student Type – careless slips

This type of student knows all the relevant skills necessary for solving problems correctly but makes the occasional careless slip. When a student produces a wrong answer to a question which requires the application of skills that the student is believed to have mastered, this may be caused by a careless slip. An example of this type of student is *sam5a:*

```
simStudents(sam5a, [[sam5a-[[makeVulgar, makeCommon],[checkAndAdd],[cancel, makeProper, makeWhole]]]]).
```

Student *sam5a* has an identical overlay model to *sam1a*. In this study, the logfiles of *sam5a* and *sam5b* were created by 'tweaking' the logfile of *sam1a* to simulate careless slips. The logfiles of *sam5c* and *sam5d* were created by 'tweaking' the logfiles of *sam1c* and *sam2b* respectively.

6.4 Generating Logfiles

The next step in the evaluation process is to generate logfiles which record the attempts of each simulated student at all the problems in the problems bank. A total of 68 problems were created, as shown in Appendix H, with the predicate, *question(N, C_p, T, P, A_p)*, that was described in Section 5.5.2.

The *question/5* predicate stores not only the questions but also the assessor's solutions to the questions, in terms of the combinations of skills used (C_p) and the final correct answers (A_p).

Each of the 28 students solves all 68 problems and their logfiles are contained in Appendix I. A fragment of the logfile of *sam2f* is given in Figure 25. Student *sam2f*, as we recall, has gaps in his knowledge and only knows the *makeCommon* and *checkAndAdd* skills.

```
simStudents([[sam2f-[[makeCommon],[checkAndAdd],[]]]]).

% a     makeVulgar
% b     makeCommon
% c     checkAndAdd
% d     cancel
% e     makeProper
% f     makeWhole

sam(sam2f,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam2f,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam2f,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam2f,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam2f,[b,c],q_bc_3,fr(5/8,1/6),fr(38/48),no).
sam(sam2f,[c],q_cd_1,fr(4/9,2/9),fr(6/9),no).
sam(sam2f,[c],q_cd_2,fr(12/64,4/64),fr(16/64),no).
sam(sam2f,[c],q_cd_3,fr(9/24,3/24),fr(12/24),no).
sam(sam2f,[c],q_cd_4,fr(9/16,3/16),fr(12/16),no).
sam(sam2f,[c],q_cd_5,fr(1/6,1/6),fr(2/6),no).
sam(sam2f,[c],q_cd_6,fr(3/8,1/8),fr(4/8),no).
sam(sam2f,[c],q_ce_1,fr(5/7,6/7),fr(11/7),no).
sam(sam2f,[c],q_ce_2,fr(4/7,8/7),fr(12/7),no).
sam(sam2f,[c],q_ce_3,fr(8/5,6/5),fr(14/5),no).
```

Figure 25. Sample of a generated logfile

In the sample, it can be seen that the student's attempt at each problem is represented by the predicate:

$$sam(S, C_s, T, P, A_s, V)$$

where S is the simulated student who applied a combination of skills, C_s, in order to solve problem P of type T. The student's final answer, A_s, is evaluated against A_p, the assessor's answer to the same problem. A verdict, V, is returned where '*ok*' means the student has provided a correct answer and '*no*' means the answer was wrong. The combination of skills, C_s, is the student's solution path which shows the set of skills he has exercised in his attempt to solve problem *P*. It may be different from C_p, the assessor's combination of skills used for the same problem.

6.5 Running the XP and ST Assessors

The generated logfiles are passed to the ST and XP assessors. ST needs information from the logfiles on each student's attempt at every problem in the question bank, while XP requires information on each student's attempts on selected problems only. The acquired information is used to infer an overlay student model and a student interaction history.

It must be noted that for each simulated student, ST and XP have access to the S, T, P, A_s and V values of the $sam(S, C_s, T, P, A_s, V)$ predicate described earlier, but not the following pieces of information:

- ***The overlay or buggy model.*** This is the L_s variable of the $simStudents(S, [S\text{-}L_s])$ predicate described in Section 6.3.

- ***The solution path or combination of skills exercised for each problem.*** This is the C_s variable of the $sam(S, C_s, T, P, A_s, V)$ predicate. This simulates the setting that the assessor has no access to the student's intermediate steps during problem solving. Whenever a student successfully solves a problem, the assessor will assume that the student has applied all the skills in the combination, C_p, which is the assessor's combination of skills used to solve the same problem.

Figure 26 are the results from running XP for *sam2e,* using information from the generated logfiles. The results of all 28 simulated students from running XP and ST are compiled in Appendices J and K respectively.

```
% XP ADAPTIVE TEST output xp_2e

% a         makeVulgar
% b         makeCommon
% c         checkAndAdd
% d         cancel
% e         makeProper
% f         makeWhole

Student =
[[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]


Selected Node : 4

Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).


Selected Node : 3

Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).


% Summary - XP1 ADAPTIVE TEST output

problems_presented(11,68).
opportunities_presented([(a,4),(b,5),(c,11),(d,7),(e,8),(f,4)]).
opportunities_correctly_applied([(a,3),(b,4),(c,7),(d,4),(e,6),(f,0)]).
```

Figure 26. Running XP on Student *sam2e*

The following pieces of information can be obtained from running the assessors:

- ***Student's attempts at problems***

  This is a record of the student's attempt at each question and is represented by the *visited/5* predicate, as described in Section 5.6.

- ***Number of problems presented***

  This is the number of problems presented to the student out a total number of possible problems. It is represented as the *problems_presented(X,Y)* predicate, where *X* is the number of problems presented during the test and *Y* is the total number of problems in the problems bank. As is typical with computerised adaptive testing, the XP assessor will always present fewer questions than ST. In the above example, XP presented *sam2e* with 11 problems, that is, 57 problems less than ST.

- ***Number of opportunities presented***

  This is the number of opportunities presented to the student in which a particular skill can be exercised. It is represented by the *opportunities_presented(P)* predicate where *P* contains a list of skills with the corresponding number of opportunities each skill could be applied.

- ***Number of opportunities correctly applied***

  This is the number of opportunities a particular skill appears to be correctly applied or exercised by a student. It is represented by the *opportunities_correctly_applied(A)* predicate where *A* contains a list of skills with the corresponding number of opportunities each skill is believed to be correctly applied. It represents the inferred overlay student model.

- ***Progression of Difficulty Level***

  The *visited/5* clauses show the progress of the student from one level of difficulty to another. In the example, *sam2e* started at level 4 and proceeded to level 3 before the test stopped. The student did not proceed to a level lower than 3 because he produced more correct answers than wrong ones at current level 3.

- ***Highest Level of Difficulty***

  A student's level of attainment, in terms of problem difficulty, is demonstrated by his ability to solve problems of the highest difficulty level possible. In the illustration given in Figure 26, *sam2e* has demonstrated that he is able to solve problems requiring the exercise of three skills. This information is useful for remedial purposes and can also be used to select the starting point if the test is administered again.

6.6 Comparing XP and ST Assessors

The results from running the assessors were that every skill had a numeric value. In broad terms, if the assessor was accurate, then each mastered skill of the simulated student should have a value greater than 1 while each unmastered skill should be assigned the value 0. The inferred overlay student model for each simulated student can be compared with the corresponding overlay, and not buggy, knowledge of the student. Information on the former is given by the predicate *opportunities_correctly_applied(A),* as described in the previous section, while information on the latter is given by the predicate *simStudents(S, [S-L_s]),* as described in Section 6.3 and Appendix G.

As an example, consider the results of running XP for student *sam2e,* as shown in given in Figure 26, where the value of *S-L* of *simStudents(S, [S-L_s])* is:

```
sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]
```

or

```
sam2e-[[a,b],[c],[d,e]]
```

and the inferred overlay model is given as:

```
opportunities_correctly_applied([(a,3),(b,4),(c,7),(d,4),(e,6),(f,0)]).
```

XP is deemed accurate as it successfully identified the mastered skills of *sam2e* as *a,b,c,d,e*, where all the values of the skills are greater than 1, and the unmastered skill as *f*, whose value is 0.

For a more detailed measure of performance, three measures are used:

- ***Accuracy on mastered skills***. This measures how good the assessor is at identifying the simulated student's mastered skills. It equals the number of times the mastered skills are correctly predicted divided by the number of opportunities those skills are presented. A mastered skill is correctly predicted if it is *correctly* applied or exercised by the student.

- ***Accuracy on unmastered skills***. This measures how good the assessor is at identifying the simulated student's unmastered skills. It equals the number of times the unmastered skills are predicted correctly divided by the number of opportunities those skills are presented. An unmastered skill is correctly predicted if the student wrongly exercises it.

- ***Overall Accuracy of Assessor***. This measures how good the assessor is at identifying the simulated student's mastered and unmastered skills. It equals the sum of the times the mastered and unmastered skills are correctly predicted divided by the sum of the opportunities these skills are presented.

As an example, consider the two samples given in Table 2 and Table 3 for student *sam2f* who has gaps in his knowledge of the subject domain. His mastered skills are the ones that are shaded. These samples are tabulated from the results after running XP and ST, as discussed in the previous section and given in Appendices J and K. The tabulated results for other simulated students, after running XP and ST, are compiled in Appendix O.

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	1	3	-	0.75
b. makeCommon	37	6	1	5	0.17	-
c. checkAndAdd	68	15	3	12	0.20	-
d. cancel	44	8	0	8	-	1.00
e. makeProper	45	9	1	8	-	0.89
f. makeWhole	11	4	0	4	-	1.00
Total:	219	46	6	40		
Average:					0.18	0.91

Table 2. Tabulated Results of *sam2f* after running XP

Skills	Total no.of opportunities	No. of opportunities presented	No. of times correctly answered	No. of times wrongly answered	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	4	10	-	0.71
b. makeCommon	37	37	3	34	0.08	-
c. checkAndAdd	68	68	8	60	0.12	-
d. cancel	44	44	0	44	-	1.00
e. makeProper	45	45	4	41	-	0.91
f. makeWhole	11	11	0	11	-	1.00
Total:	219	219	19	200		
Average:					0.10	0.91

Table 3. Tabulated Results of *sam2f* after running ST

The second column '*Total no. of opportunities*' represents the maximum number of times each skill can be presented. It is the accumulation of the number the times each skill appears in the skills combination of every question in the question bank given in Appendix H. In the above example, the total number of opportunities for all six skills to be presented is 219. XP requires only 46 opportunities, or 21%, of a total of 219 opportunities while ST requires all 219 opportunities.

The accuracy of XP in identifying a mastered skill of *sam2f*, say *makeCommon*, is 0.17, and is calculated as the number of opportunities the skill is correctly applied (column 4) divided by the number of opportunities the skill is presented (column 3). The accuracy of XP in identifying a unmastered skill of *sam2f*, say *makeProper*, is 0.89, and is calculated as the number of opportunities the skill is wrongly applied (column 5) divided by the number of opportunities the skill is presented (column 3).

In studying the results of *sam2f* above, ST and XP performed well in assessing the unmastered skills of *sam2f* but were not accurate in their prediction of the student's mastered skills. Upon closer observation, XP performed marginally better than ST, despite having presented far fewer problems than ST.

There is, however, a limitation to this method of measurement. There may an occasion when a student has a mastered skill but may have produced an incorrect answer to a problem which had required the application of the mastered skill. This could be caused by the presence of one or more unmastered skills which led to the eventual incorrectness of the final answer. This is not obvious as the assessor only evaluates final answers as correct or wrong; it has no access to the intermediate steps of the student's solution. As an example, consider the *makeCommon* skill in Table 2. It can be seen that although this skill is one which is mastered by the student, given a total of 6 opportunities, the student is seen to have correctly applied this skill only once. This could be caused by the possibility that the problems presented may require the application of unmastered skills, such as *makeVulgar, cancel, makeProper* or *makeWhole.*

The tabulated results of Appendix O which present the performance of XP and ST in assessing the different types of students can also be shown as bar charts. Figure 27 compares XP and ST in their accuracy in assessing mastered skills, while Figure 28 compares their performance in assessing unmastered skills. In Figure 28, information on *Sam1* type of students is absent. This is because this type of students does not have unmastered skills. The overall performance of XP and ST in assessing both mastered and unmastered skills is shown in Figure 29.

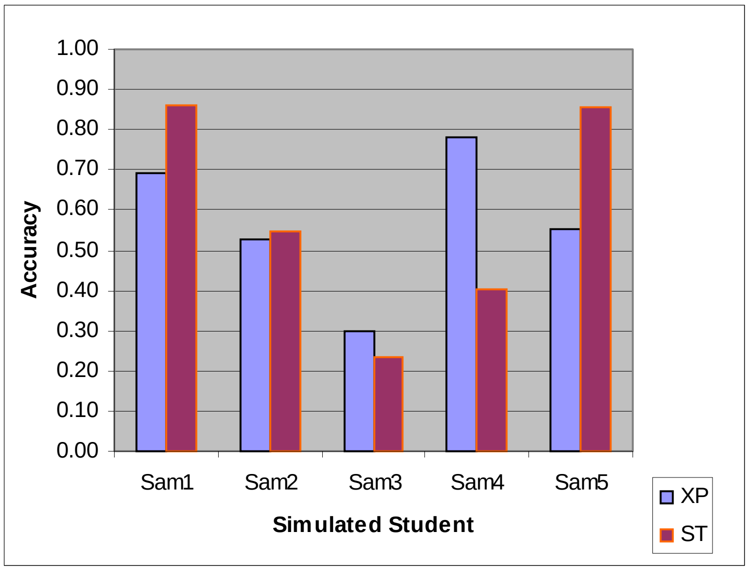

Figure 27. Comparing XP and ST – Accuracy of Mastered Skills

In assessing mastered skills, as shown in Figure 27, ST fared considerably better than XP in inferring the mastered skills of *Sam1* type who knows all the skills and *Sam5* type with careless slips. There are only marginal differences in their accuracy in assessing *Sam2* type who has gaps and *Sam3* type with malrules. XP performed considerably better than ST in assessing *Sam4* type of students with lucky guesses.

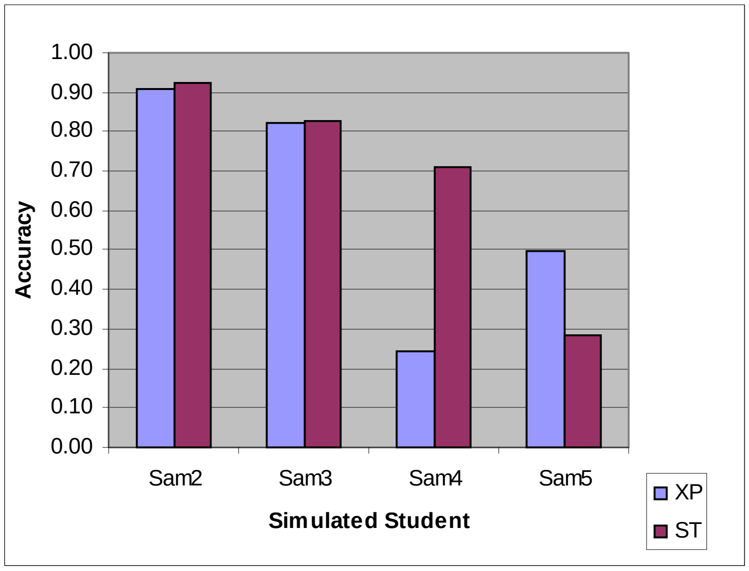

Figure 28. Comparing XP and ST – Accuracy of Unmastered Skills

In assessing unmastered skills, as shown in Figure 28, XP and ST fared equally well for *Sam2* and *Sam3* types. ST was better than XP in its prediction of unmastered skills of *Sam4* type but fared worse than XP in assessing *Sam5* type of students.

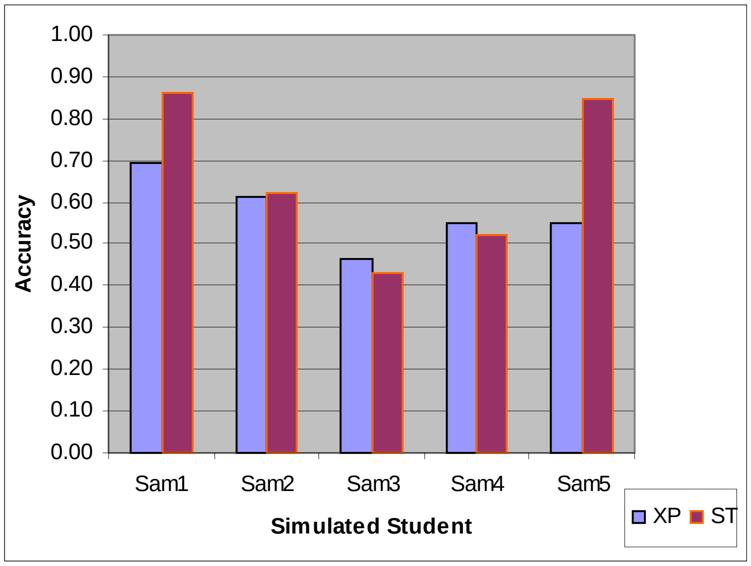

Figure 29. Comparing XP and ST – Overall Accuracy

In terms of overall accuracy, as shown in Figure 29, XP is better in assessing the mastered and unmastered skills of students who knew the skills (*Sam1*) or have gaps in their knowledge (*Sam2*) than it is with students with noisy data such as malrules, lucky guesses or careless slips. ST performed best in its assessment of students who knew all the skills (*Sam1) and* those with careless slips (*Sam5*). It performed only marginally better than XP in assessing students with gaps in their knowledge (*Sam2*) but was marginally worse than XP in assessing students with malrules (*Sam3*) and those with lucky guesses (*Sam4*).

A possible reason for XP's low accuracy in assessing students with noisy data is that it is not capable of detecting inconsistencies in student answers. It is relatively easier for a human assessor to detect such inconsistencies, as discussed in the DSA experiment of Section 4.2.1, especially when the assessor has access to student problem solving steps as well as final answers. As an example, consider student *sam3a* who has a malrule, *mg* or *malCancel,* which is a misconception of the *cancel* skill. His attempts at some questions in which he exercised the malrule are given below:

```
sam(sam3a,[c,mg],q_cd_2,fr(12/64,4/64),fr(1/4),ok).
sam(sam3a,[c,mg],q_cd_3,fr(9/24,3/24),fr(1/4),no).
sam(sam3a,[c,mg],q_cd_4,fr(9/16,3/16),fr(2/6),no).
```

The student used *mg* to solve question *q_cd_2* successfully, but not the next two questions. This malrule was described by Borasi (1994) in Section 4.2.1.4 where the student eliminates similar digits from the numerator and denominator. The addition of the two fractions in the first question gives 16/64 and the student applied the malrule *mg* which led to the correct answer, 1/4. However, when he attempted to apply the same malrule to next two questions, his answers were incorrect. The application of the malrule is 'masked' in the first answer but not in the second or the third. As the XP assessor has no memory of previously answered questions, it was not able to detect such inconsistencies.

The question that arises is whether XP can detect inconsistencies without performance assessment or model tracing. This may be achieved with a procedure that reviews or inspects all the final answers of the student at the end of the test in order to check for inconsistencies. For example, if a student consistently produces wrong answers to questions which require the application of a certain skill, then it can be inferred that the student does not know that certain skill. However, if his behaviour is inconsistent, then this presents an avenue for checking for the possibility of noisy data such as malrule application, lucky guesses or careless slips. The characteristics of each possibility must be known precisely in order to distinguish one from the other.

Inconsistent behaviour could also be caused by the student using a different solution path or a different set of skills to the one predicted by the assessor. This possibility could be reduced if each problem in the problems bank is saddled, wherever possible, with more than one possible solution path or combination of skills. The assessor can then be equipped with a heuristic function which checks for all possible causes for inconsistent behaviour and settles on the most probable one.

Another shortcoming of XP arises because it has been implemented for evaluation by simulation. If it were used in classroom tests, because it would present identical tests to two, it is hard to distinguish a good student from one who copies from him. To resolve this, problems could be selected at random.

6.7 Varying the Parameters of XP

Many variations of XP could be studied. The previous section described the performance of XP based on the values of its parameters identified in Section 5.4.1. This section investigates the effects of varying some parameters of XP on its accuracy in assessing the mastered and unmastered skills of the different types of students. The selected parameters are:

a. Number of problems per combination of skills

XP currently requires the presentation of only one problem per skills combination. A variation is to increase this to two problems per combination.

b. Progression of Difficulty Level

The progression from one level of difficulty to another is described in Section 5.4.1. As an example, consider the results of *sam1a* after running XP (Appendix J). The student started the test at difficulty level 4, and as he consistently produced correct answers, he progressed to level 5 and then to level 6, although at this level, there were no problems in the database that matched a six-skill combination. In the case of *sam2f* who knew only two out of six skills and produced more incorrect answers than correct ones, progress was from level 4 to level 3, then to level 2 and level 1. A variation is to calculate the midpoint by *rounding down*, instead of rounding up, to the nearest integer. For a good student like *sam1a*, progress will then be from level 3 to level 5 and then level 6. For a weak student like *sam2f*, progress will be then from level 3 to level 1. If at level 1, *sam2f* produces more correct answers than wrong ones, he will proceed to level 2 where he will be presented with problems requiring the exercise of two skills and the test stops.

c. Threshold level of success

A student progresses from one difficulty level to another based on how he fared at the current level. For example, if 10 problems were presented and 6 were answered correctly, the student progresses to a higher level of difficulty. A variation is to raise this threshold to 0.75, which means that the student needs to answer correctly at least 8 problems out of 10 if he is to be allowed to progress to a higher difficulty level; otherwise he will be presented with easier problems at a lower difficulty level.

The next section discusses the results from running variations of the XP assessor.

6.8 Running Variations of XP

Variations of XP were created, namely XP1, XP2 and XP3, based on changes to the three parameters suggested in the previous section. XP1 incorporates a variation to the first parameter that is presenting two problems per combination of skills. XP2 incorporates variations to the first and second parameters, where the latter involves recalculating the next difficulty level as rounding down to the nearest integer, the midpoint between two difficulty levels. XP3 represents variations to all three parameters, where the last parameter involves raising the threshold of success to 0.75. The three newly created versions of XP were run and the results of selected students are tabulated in Appendices L, M and N.

A summary of results, which compares the performance of the five assessors, is given in Appendix P. The results can also be represented as bar charts. Figure 30 shows the chart for overall accuracy at assessing mastered and unmastered skills.

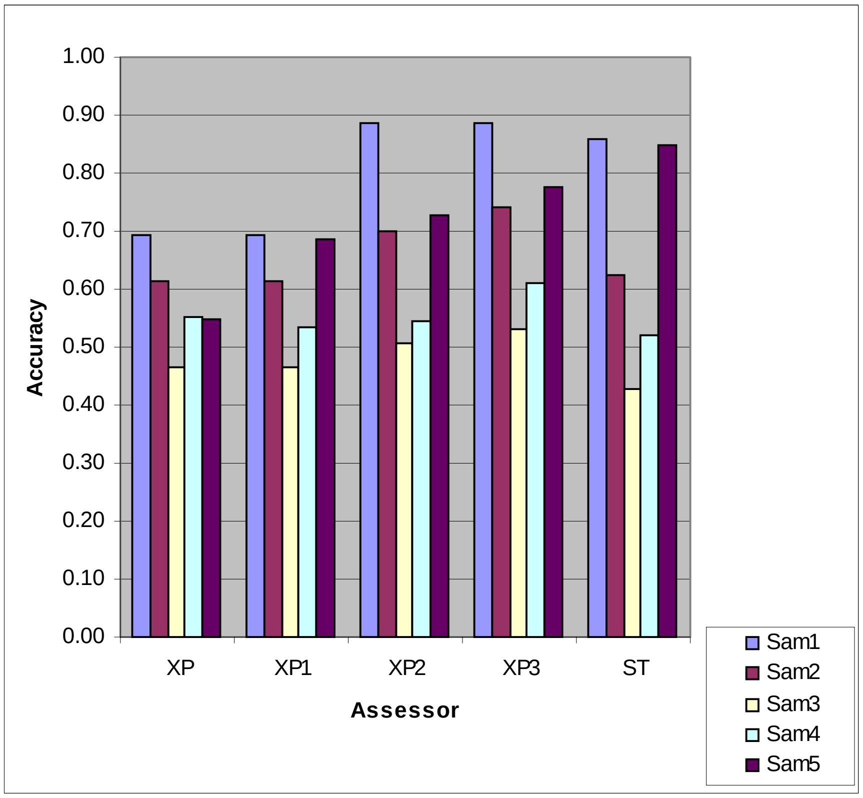

Figure 30. Comparing Assessors – Overall Accuracy

The following observations were made in relation to the varying parameters introduced in the previous section:

- **Accuracy is not proportional to the number of questions presented**
  An increase in the number of questions presented, from 1 per skills combination to 2, does not seem to have an effect on accuracy. This can be inferred by comparing XP and XP1. For example, apart from *Sam5* students, there was no evidence of increased accuracy with the doubling of questions posed to the student.

- **Increased accuracy with changing progression of difficulty level**
  There was an improvement in accuracy with the introduction of a different navigation of test difficulty. This is evident by comparing the performance of XP2 to XP1.

- **Marginal increase in accuracy with a higher threshold of success**
  Performance only improved marginally with the introduction of a higher threshold of success which dictated progression to another level of difficulty. This can be seen by comparing XP2 and XP3. With *Sam1* type, there was no increase in accuracy.

Figure 31 compares the performance of the different assessors in their prediction of mastered skills. Apart from increased accuracy for *Sam5* students with careless slips, performance did not improve with the presentation of more problems, from XP to XP1. In fact, accuracy depreciated significantly in assessing the mastered skills of *Sam4* type of students with lucky guesses. There is however a significant increase in accuracy with the introduction of a changed progression of difficulty level. There is a marginal increase in accuracy for most assessors with the introduction of a higher threshold of success. ST fared well in assessing the mastered skills of *Sam1* and *Sam5* types of students but it performed worse than its counterparts in assessing the mastered skills of the other types of students.

Figure 32 compares the performance of the different assessors in their prediction of unmastered skills. As can be seen from the figure, data for *Sam1* type is absent and this is because this type of students has no unmastered skills. On the whole, apart from its assessment of *Sam5* type, ST fared better than the other assessors in assessing unmastered skills. The introduction of the varying parameters seems to have an adverse effect on the performance of the assessors.

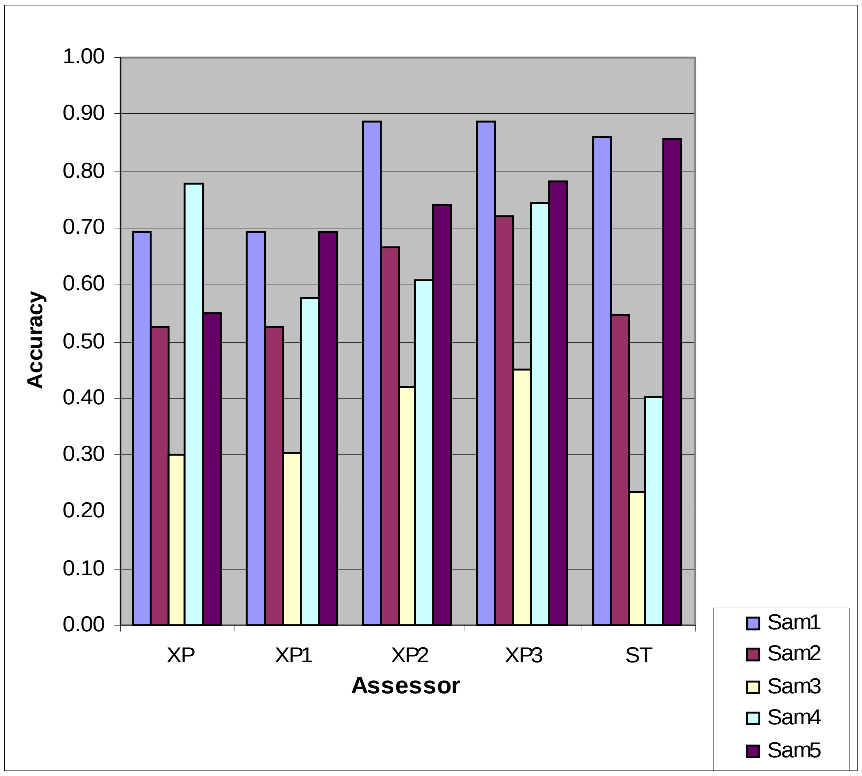

Figure 31. Comparing Assessors - Accuracy of Mastered Skills

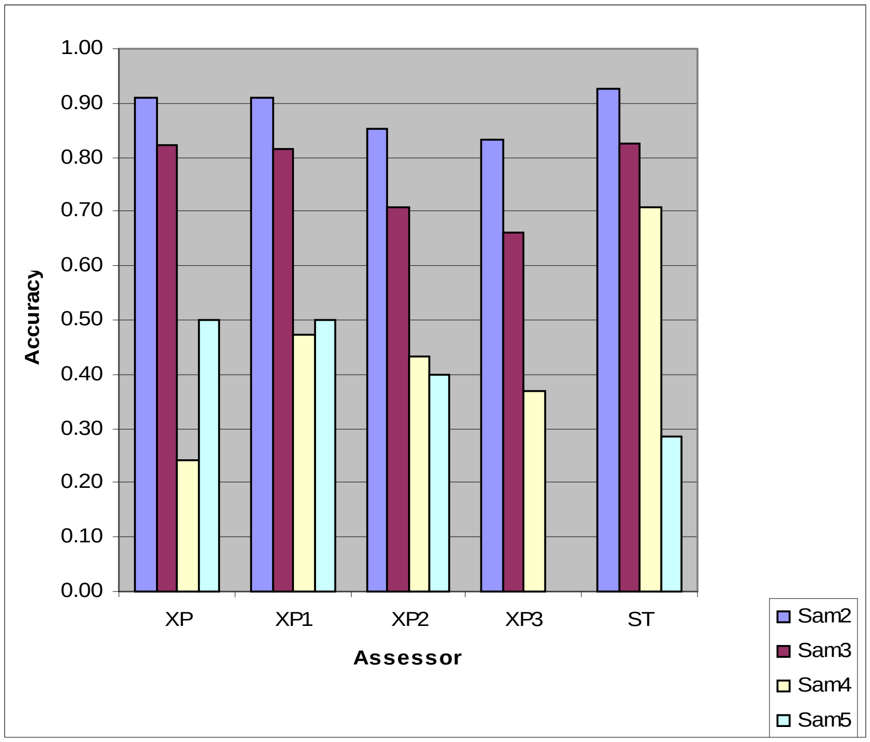

Figure 32. Comparing Assessors – Accuracy of Unmastered Skills

6.9 Conclusion

This chapter presented an evaluation strategy as a means of measuring the performance of an adaptive testing strategy, XP, under varying conditions provided through the use of simulated students. Three steps were identified in the evaluation and these led to an inferred overlay student model and a student interaction history for each simulated student. Each inferred overlay student model is compared with the overlay model of a simulated student. A more detailed measure of performance was proposed which measured the accuracy of the assessor in identifying mastered and unmastered skills. The results were compared with those of a sequential testing strategy, ST.

The XP assessor was good at differentiating a student who knows all the relevant skills in the subject domain from one who had gaps in his knowledge, but it was not good at making fine distinctions between students with mal rules, lucky guesses or careless slips.

Variations of the assessment were studied. The values of three numerical parameters of XP were readjusted in order to detect improvement in accuracy. It was observed that an increase in the number of questions presented and an increase in threshold level of success had little impact while changing the navigation of test difficulty improved accuracy.

An important finding is that adaptive testing can perform as well, and in some cases even better, than sequential testing. No single assessor was found to be good at inferring the mastered and unmastered skills of all student types. On the whole, XP3 fared the best in assessing all student types, except *Sam5* type of students with careless slips, where it came second place to ST. A possible scope for expansion involves further tuning XP3 by varying a variety of parameters and revaluating its performance.

Chapter 7.

Conclusions

This thesis has discussed computerised adaptive testing in the context of Intelligent Tutoring Systems and student modelling. It has explored the scope for exploiting the Expert Systems technique of knowledge acquisition in the construction of adaptive tests. In particular, the thesis shows how software-aided knowledge acquisition can make a contribution to syllabus description and to determining the sequence in which questions should be presented to students. The thesis also discusses the delivery of adaptive tests.

This chapter presents a summary of this work. It highlights the main contributions of the thesis and discusses the scope for future work.

7.1 Summary

Chapter 1 addressed the motivation and aim of the research. It presented the idea that teachers, working in a small and familiar domain, may have good adaptive testing strategies for assessing the student's state of knowledge in a subject domain. It proposed an expert emulation approach to designing and constructing adaptive tests with the ultimate aim of incorporating the results of elicitation in a student modelling architecture called SKATE.

Student modelling and student testing have similar aims and may use similar techniques. Chapter 2 presented a review of literature on student modelling in intelligent tutoring systems. It examined the different techniques and challenges of student modelling and discussed the relationship of adaptive testing to student modelling. The implications of the review on the design of SKATE are highlighted especially in the construction of the domain and the student model. It further discussed computerised adaptive testing in the light of other testing strategies and presented two common approaches in adaptive test design and construction, namely the Item Response Theory and the Knowledge Space Theory. The former seems an extremely effective means of conducting summative testing but has little application to formative testing while the latter concerns itself more with formative testing. The implications of the review on the design of SKATE are highlighted especially in the structuring of the domain and in the issue of problem progression.

The first part of Chapter 3 is devoted to a discussion of preliminary topics to provide a foundation for knowledge acquisition work described in subsequent chapters. It introduced the problem domain of elementary arithmetic with fraction additions and discussed the teaching role of the teacher who is the “expert” in the expert systems aspect of the thesis. It described an observational study of the expert’s assessment techniques which paid particular attention to the issue of “adaptability” and highlighted the possibility of using knowledge acquisition support software based on constraint logic programming, clp(FD). The second part of Chapter 3 described the results of elicitation in the construction of a problem domain which describes a test syllabus. This work discussed the successful application of clp(FD) as a tool for knowledge elicitation, knowledge representation and rapid prototyping. The description of a problem domain in terms of constraint logic programs allows it to be *executable* and this facilitates the evaluation of student answers and the generation of problems for use during knowledge acquisition and test delivery.

Chapter 4 discussed experiments in creating a student model and problem progression strategy in adaptive testing. In determining the contents of a student model for adaptive testing, findings of two experiments were used to aid in the decision where the student model will maintain domain-specific information about the student as an overlay model and an interaction history module. The clp(FD) representation of the overlay model meant that it is executable and this is useful for predicting student performance in problem solving and for generating problems to aid remediation. The crucial element in adaptive testing is the selection and progression of questions and the determination of the stopping place. Chapter 4 further described two experiments which elicited the problem progression strategy of the expert. The first method involves a computer-aided query procedure which systematically elicits the task of problem sequencing. The use of such software makes the potentially tedious process acceptable to experts. The result is a binary-tree algorithm called BT. The second method presents an alternative strategy of problem progression which is based on the expert's measure of problem difficulty. This resulted in the design of a skills-based algorithm called XP.

Chapter 5 described the student modelling architecture, SKATE. It is based on the information acquired from an expert teacher of elementary arithmetic. The underlying domain knowledge relates to a particular view of the topic by a particular person. The approach to adaptability developed in SKATE is not the only technique used by the domain expert. It has the advantage that it can be used as a basis for a series of experiments that charts the consequence of a range of minor variations. An account is given of such a range of experiments using a simulation of student performance to follow through the consequences of such changes in the next chapter.

Chapter 6 describes an evaluation strategy carried out to measure the performance of XP provided through the use of simulated students. Different types of students were simulated – students who knows all the skills, students who have gaps in their knowledge, students with malrules, students making lucky guesses and students making careless slips. The results from the evaluation were compared with those of a sequential testing strategy, ST. It was found that XP fared relatively well, despite presenting far fewer questions to the student than ST. Variations of XP were created and evaluated against XP and ST.

7.2 Publications

Several of the ideas discussed in this thesis have given rise to conference and workshop presentations. These have not been cited in the thesis but are listed in Appendix Q.

7.3 Main Contributions

This thesis takes the idea of adaptive testing, that were initiated by 'Item Response' theorists such as Lord (1980) and Weiss and Kingsbury (1984), who were psychometricians, and shows how that idea can contribute to the work of the Intelligent Tutoring Systems researchers. To do this, it was necessary to relate adaptive testing to the central concept of Intelligent Tutoring Systems, namely the student model that was introduced by John Self in 1974. It was also necessary to develop implementation techniques, since computer development and delivery are obviously essential for intelligent tutoring. Of course, psychometricians are also concerned with computing (Doignon and Falmagne, 1985), and indeed computers are the obvious medium for the delivery of adaptive tests. However, the focus of the work of psychometricians in this area is on the development of mass tests where reliable statistical quality control assurance can be provided (Weiss and Kingsbury, 1984). In contrast, the emphasis of the Intelligent Tutoring community has been on the individual student, and this thesis has thus been concerned with the economical way of developing and delivering adaptive tests for individuals and small groups of students.

The main contribution of the thesis is embedded in the design of a student modelling architecture called SKATE, which is intended for the development, delivery and evaluation of adaptive tests. SKATE is composed of many of the components that are to be expected in an Intelligent Tutoring System, but with the pedagogic or tutoring model replaced by a test delivery model. The experimental problem domain for SKATE had a considerable influence on its design. By choosing to work with the algebra of integer fractions, it was necessary to find computationally flexible ways of representing arithmetic problems which were limited to a subset that teachers, tutors and text book authors think suitable for their students. Since Constraint Logic Programming (CLP) permits a statement of integer domains, and since it interfaces seamlessly with Prolog, it is an obvious choice for representing the test material. But the representation of the problem domain in an intelligent tutoring system influences most other components, and so CLP affected the design on most components in SKATE. In addition to the domain model, these components are the student model and the test delivery model. They are discussed in the following sections and a critical assessment is made of the contribution to the current state of research in adaptive testing and intelligent tutoring systems, and the relationship of SKATE to the broader question of 'learning'.

7.3.1 The Domain Model

The use of constraints in domain models is not new. Ohlsson (1994) and Mitrovic (1998) used constraints to model an envelope of possibilities, which is actions or interpretations, in a particular domain that can be used to map a student's performance onto a range of pedagogic corrective strategies. In the SKATE model, constraints are used to model problems and not student behaviours. Further, there is no place in an adaptive test for corrective strategies. The action that follows either success or failure with a test item is determined by a separate testing strategy and is independent of the details of the way in which a student completes a test item.

However, even though the representation of the testing strategy does not make use of CLP, it was used as a means of facilitating the knowledge elicitation process used to arrive at a strategy. Section 3.6 describes an interactive process that relies on the generation of sample problems coded as CLP program fragments to elicit an order in which problems should be presented. The feature of CLP that is exploited here is its ability to simultaneously represent both a narrowly defined class of problems and particular examples of that class.

CLP does not compete directly with other techniques used for domain representation in Intelligent Tutoring Systems. It is an extension of the expressive power of logic as well as an algorithm for solving constraint problems. So it may be used to facilitate and extend a representation technique, or it may find a place in tandem with some other techniques. The two standard techniques of Intelligent Tutoring Systems are the mal-rule or buggy technique used to represent student misconception, and the automatic problem solving technique used to represent a student's search path from problem presentation to solution. Both of these combine elements of a student model as well as the domain model, and are discussed below.

Brown and Burton (1978) use mal rules in their classic DEBUGGY system that is discussed in Section 2.4 and which, like SKATE, deals with arithmetic problems. Its domain representation could be extended by CLP. CLP facilitates the representation of discrete domains, and operation that can be performed on them, and it is as easy to represent the use of an incorrect, or buggy operator, as a correct one. Consider the example used by Ohlsson (1994), of the elementary mistake of summing fractions by summing the numerators to give the numerator of the result, and summing the denominators to give the denominator of the result. This can be represented by the following fragment:

```
BuggyAdd(N1,D1,N2,D2,N3,D3) :-

domain([N1,D1,N2,D2,N3,D3],1,9),         % Single digit integers
N1 #<  D1,                               % First operand - proper fraction
N2 #<  D2,                               % Second operand - proper fraction
N3 #= N1 + N2,                           % Sum the numerators
D3 #= D1 + D2.                           % Sum the denominators
```

This not only exactly models the “buggy” operator, but can be solved to deliver erroneous summations.

The widely used alternative approach to domain and student modelling, usually referred to as the Machine Learning Approach (Section 2.4.8), does not stand to benefit from the use of CLP in the same direct way but they may be used together. This approach relies on the exact modelling of one or more solution paths. Although it is possible to exercise some control over CLP’s constraint satisfaction algorithms, there is no mechanism for controlling this in sufficient detail to model a human protocol. So in SKATE, a backward-chaining problem solver is used to analyse problems in terms of problem-solving skills needed for a solution. This information is used for problem assessment as part of one of the delivery algorithms (Section 5.5.2). The strength of CLP lies in its ability to model problems, rather than problem domains, where the solution or sets of partial solutions are more significant than the solution path. Its potential for use in mal rules or buggy rules though not studied in this thesis should be worth further investigation.

7.3.2 The Student Model

The student model used in SKATE records a student’s progress through a series of test items. Such a model, which relies on the representations in the domain model, is called an overlay model. It records a student’s success or failure for each test item and also records those features of a test that are relevant for determining the “next” test item to be presented. Information about a student’s competence has to be inferred from this record of results, so competence or the lack of competence is characterised in terms of the characteristics of test items.

From the adaptive test perspective, the crucial decision is always what test item should be presented next. Two aspects need to be discussed. One is the reliability of item results; the other is determining which test item should be delivered next in order for the maximum information about the student’s ability to be gathered with the smallest number of test items. It is in answering the former question that other researchers have used Bayesian statistics; see for example the ANDES system (VanLehn and Niu, 2001). This approach is compatible with the CLP based domain model of SKATE, but would require information to indicate how to update a probability hypothesis in the light of the result of a test item. This information is not available within SKATE, and decisions about the progression through test items are dealt within the delivery module, which is discussed below.

7.3.3 The Test Delivery Model

There are two broad approaches that have been used for determining the order in which the questions that constitute an adaptive test are asked. These are the statistical approach, such as Item Response Theory (Wainer and Mislevy, 1990) and the Knowledge Space approach (Doignon and Falmagne, 1985). These are discussed in Chapter 2. The statistical approach studies patterns of behaviour of population samples in order to discover the patterns of co-occurrence of success or failure. The Knowledge Space approach relies on a semantic analysis of a problem domain. This can be carried out directly, by looking for a semantic ordering of prerequisites (Dowling and Kaluscha, 1995) or by the use of knowledge acquisition from educational experts (Koppen, 1993; Kambouri et al. 1994).

This thesis uses two variants of the knowledge acquisition approach. Section 4.5 discusses a technique for knowledge acquisition directed to determining a specified sequence of classes of test items, and Section 4.6 discusses a more abstract approach. The former is only suitable for small tests, since the knowledge acquisition process is quickly swamped by combinatorial explosion of possible test paths. The second approach, the XP strategy, is independent of the size of the test. The XP strategy focuses on the mastery of specific “micro-skills” needed to solve particular problems. Its limitation is that it treats all identifiable skills as equally significant and equally prone to error. However, the strategy is independent of the number of number of problem classes, and so scales without problem. The worse case delivery performance is proportional to the log to base two of the number of problem class clusters used.

There is a growing body of research in this area. McCalla et al. (1992) have studied the issues of levels of detail, or granularity; Hirashima et al. (1996) have used a notion of the simplification of problems, though this has been in the context of tutoring rather than testing. Work that has been carried out with the intent of developing tests has focused on the representation of problems. Work carried out that has been concerned with tutoring has not shared this focus, and is more concerned with sequencing the introduction of skills and concepts. This is an area in which more research is needed in order to improve the accuracy and coverage of adaptive tests.

When XP was evaluated using simulated students, its assessment of different types of students was found to be comparable to that of sequential testing (Section 6.6) and it was found that the accuracy of XP could be further fine-tuned by varying its parameters (Sections 5.4.1 and 6.7). The problem with using simulated students is that the simulation is based on the same “knowledge base” as the adaptive test or intelligent tutoring system that is being evaluated. Exactly the same assumptions that underlie the adaptive test or intelligent tutoring system also inform the construction of the simulated students. This clearly limits what can be learned from simulation experiments. But as mentioned above, simulation may find a role in “fine tuning” the parameters of a test, or estimating the number of interactions required when using an intelligent tutoring system.

7.3.4 Learning

SKATE has no assumptions about learning: it is only concerned with testing. However, the interpretation of a test result is sensitive to assumptions about:

- whether or not learning takes place during the testing process;
- the ambiguity that arises when there are several ways of solving a problem; and,
- the complexity of assessing a skill use, when the skill can only be demonstrated in conjunction with other skills.

However, it is not just SKATE that has this sort of problem. Van Lehn and Niu's BBN system (2001), bug libraries, (Brown and Burton, 1978) and machine learning techniques such as ACM (Langley and Ohlsson, 1984) all suffer from *radical strategy variability* (Section 2.4.7), since a student can have several strategies in use at any moment in time, and may switch between them on a problem-by-problem basis.

SKATE presents a quick means of assessment that helps the student identify straightaway the skills that he needs to reinforce learning on, and focus on learning this aspect. It assumes that if a student has any unmastered skill, this will surface by constant questioning. The careful placing of questions will minimise the possibility of student using different solutions and not necessarily those skills. In this way, SKATE supports many learning theories. Perhaps the closest is the Information Processing learning theory (Miller 1956). There are two concepts in this learning theory framework – chunking and information processing. Chunking is first advocated where the student holds several pieces or chunks of information in his short-term memory (encoding or retention). In the current domain of study, a chunk can be a problem solving skill. Next, the student is presented with a problem. At this stage, information processing takes place, where the student retrieves relevant chunks and applies one or more skills in order to solve the problem.

Assessment is, and will remain, closely associated with teaching and learning. Evaluating the progress of a student is a vital part of both tutoring and teaching systems. The state of knowledge of a student in a subject domain is best assessed when he is not anxious. Computerised adaptive testing promises an effective and accurate strategy. This thesis presented a feasible approach to creating small-scale tests with the hope that this software-supported technique will eventually find its way in educational and training settings in class rooms and elsewhere.

7.4 Further Work

Adaptive testing has been developed independently of tutoring systems, and it has been mainly used for large-scale summative evaluation. This thesis has been concerned with developing a range of techniques which, amongst other things, are applicable in small-scale testing. They are useful particularly for transient student populations or for students engaging in lifelong learning with gaps in their knowledge. But the same techniques are applicable as alternatives to fixed length tests in normal classroom teaching. Research needs to be undertaken, which would benefit from collaboration with textbook authors and publishers as well as with classroom teachers, to evaluate this possibility. The present work has only investigated the use of CLP to represent arithmetic problems. Though there are many potential applications in the field of mathematics and related subjects, it should be fruitful to explore the application in all those areas of management science, engineering, planning and design that have presented constraint satisfaction problems.

A second area of application is in conjunction with, or integrated with, an ITS system. Here SKATE could be used for pre-testing before proper tutoring begins, much like a pretest for SIETTE (Arroyo et al. 2001) and SMART (Shute 1995). There is interesting work to be undertaken here which would involve the integration of the testing needed for example for the detection of misconceptions on the one hand and performative competence on the other.

There is a more interesting possibility, which is suggested by the nature of CLP, whose constraint solving algorithm automatically provides a family of problems and solutions from a description of a class of problems. This is the potential for the generation of test items from a complete representation of domain of an intelligent tutoring system. This would require a pedagogic model that would provide a structure for tutorial topics, problem classes and tutorial information in an explicit fashion so that the progression problem of adaptive testing could be solved by drawing on the same material used for sequencing tutorial strategies.

Appendix A. Item Characteristic Curves

This appendix contains item characteristic curves for the 2-PL and 3-PL models for different values of *a*, *b* and *c*. Item characteristic curves were discussed in Section 2.5.1.

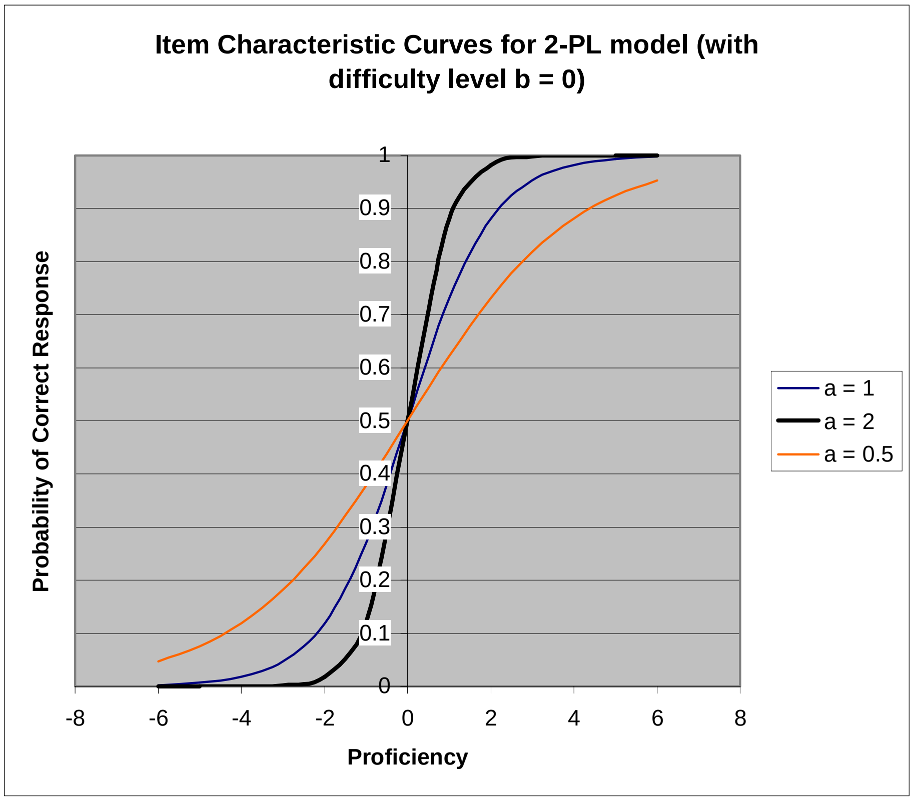

Figure 33. 2-PL Item Characteristic Curves (b=0)

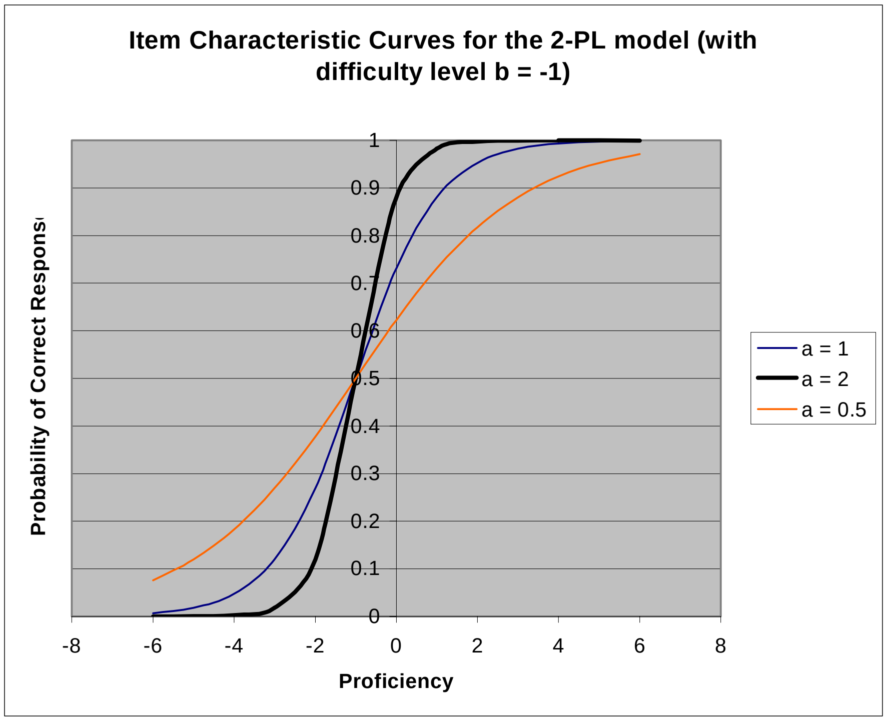

Figure 34. 2-PL Item Characteristic Curves (b=-1)

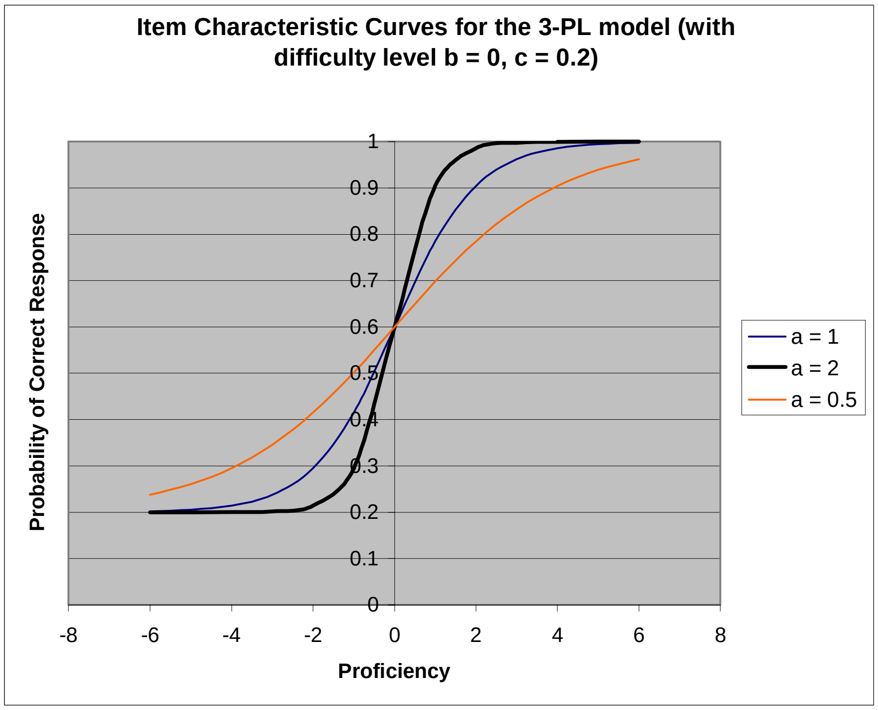

Figure 35. 2-PL Item Characteristic Curves (b=0, c=0.2)

Appendix B. Manual Adaptive Testing

This appendix contains two interactive sessions which took place between the expert and two students on a one-on-one basis. This was discussed in Section 3.2.5.

Subject 1: An 11 year old student on the UK National Curriculum, Year 6

Task/Observation	Problem posed	Student's response
Start the test Teacher's impression of student through cues: student seems confident	2 ½ + 3 ¾	Thinking aloud: ¾ = ½ + ¼, so ½ + ½ = 1. Add that to the whole numbers gives 6. The remaining fraction is ¼, so the answer is 6 ¼
The student has used a different strategy to the one predicted by the teacher. The teacher had expected the student to add ½ and ¾ to arrive at an improper fraction before converting it to a proper fraction and adding the whole number to the other whole numbers. Student used the skills *number facts, sum whole numbers and add equivalent fractions* but not the expected skills of *find common denominator, find equivalent fraction* and *make proper.* The teacher is also aware that the skill *cancel fraction* has not been tried yet. He expects student to demonstrate mastery in the skills *find common denominator*, *find equivalent fractions* and *make proper.* This influences the choice for the next problem	4/3 + 5/4 Can you break it down further?	16/12 + 15/12 = 31/12 Struggles and wrote 2 r 7 / 12 and then 27/12 and then 2 7/12
Teacher thinks: "Can the student really solve fractions with different denominators. Did he apply LCM or did he just get the product of the denominators to get a common denominator. My previous question does not reveal this, so I will pose the next one"	7/2 + 8/3 – 6/5	Struggles to get LCM for 3 numbers 2 3 5 4 6 10 6 9 15 8 12 20 10 15 25 12 18 30 14 21 35…..and said LCM = 4, then thought again

Teacher intervenes with a simpler problem	8/3 – 6/5	40/15 – 18/15 = 22/15 = 1 7/15 {solves successfully then resumes to solve previous problem} 105/30 + 80/30 – 36/30 = 149/30 = 4 29/30
Again, teacher's choice of question did not test for the ability of the student to use LCM, so another question was devised The skill *cancel fraction* has not been tested, so I will give a specific question on it Stop the test Verdict: Student has demonstrated mastery in all skills and can handle fraction addition and subtraction up to 3 operands	4/3 – 1/6 + 5/4 What if the answer was 2 3/12, can you break that down any further	{Thinks aloud} LCM = 24, now can I make it any lower…..12 16/12 – 2/12 + 15/12 = 29/12 = 2 5/12 2 ¼

Subject 2: A 10 year old student on the UK National Curriculum, Year 5

Task/Observation	Problem posed	Student's response
Start the test Teacher's impression of student through cues: student looking math anxious, therefore start test with very easy problem	2/3 + 3/4	5/7
Student has added both numerator and denominator – a clear misunderstanding of the concept of fraction addition		
Give a simpler problem	2/5 + 3/5	5/10
Student has displayed the same misconception		
Give another problem	1/4 + 2/4	3/8
Stop the test Verdict: Student has demonstrated misunderstanding of the concept of fraction addition. Remedial help should take the form of visual display e.g. pieces of a pizza, to demonstrate the idea of fractions and then fraction addition		

Appendix C. Clp(fd) Representation of Problem Classes

This appendix contains a clp(FD) representation of the problem classes for the domain of fraction additions. Each procedure can be executed to generate one or more example problem or problem instances. This was discussed in Section 3.6.1.

Question Type PT1: Add Two Proper fractions of common denominator

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt1(N1,D1,N2,D2,N,D):-
        domain([N1,D1,N2,D2], 1,9),            % Single digit integers for numerators
        domain([N,D],1,99),                    % Possible values for answer
        N1 #< D1,                              % First operand - proper fraction
        N2 #< D2,                              % Second operand - proper fraction
        D1 #= D2,                              % A common denominator
        D #= D1,                               % Same denominator in solution
        labeling([], [N1,D1,N2,D2, N,D]),      % Generate values for all variables
        \+ cancel(N1,D1,_,_),                  % First operand in simplest form
        \+ cancel(N2,D2,_,_),                  % Second operand in simplest form
        N1/D1 + N2/D2 =:= N/D.                 % Arithmetic expression
```

Question Type PT2: Add a Proper fraction and an Improper fraction of a common denominator

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt2(N1,D1,N2,D2,N,D):-
        domain([N1,D1,N2,D2], 1,9),            % Single digit integers for numerators
         domain([N,D],1,99),                   % Possible values for answer
        N1 #< D1,                              % First operand - proper fraction
        N2 #> D2,                              % Second operand - improper fraction
        D1 #= D2,                              % A common denominator
        D #= D1,                               % Same denominator in solution
         labeling([], [N1,D1,N2,D2, N,D]),     %Generate values for all variables
        \+ cancel(N1,D1,_,_),                  % First operand in simplest form
        \+ cancel(N2,D2,_,_),                  % Second operand in simplest form
        N1/D1 + N2/D2 =:= N/D.                 % Arithmetic expression
```

Question Type PT3: Add two improper fractions of a common denominator

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt3(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),                  % Single digit integers for numerators
        domain([D1,D2], 2, 9),         % Single digits, start at 2 to avoid 1 in denominator
        domain([N,D],1,99),                    % Possible values for answer
        N1 #> D1,                              % First operand - improper fraction
        N2 #> D2,                              % Second operand - improper fraction
        D1 #= D2,                              % A common denominator
        D #= D1,                               % Same denominator in solution
        labeling([], [N1,D1,N2,D2, N,D]), %Generate values for all variables
         \+ cancel(N1,D1,_,_),          % First operand in simplest form
        \+ cancel(N2,D2,_,_),           % Second operand in simplest form
        N1/D1 + N2/D2 =:= N/D.          % Arithmetic expression
```

Question Type PT4: Add two proper fractions of different denominators where the denominators are multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt4(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),               % Single digit integers for numerators
        domain([D1,D2], 2, 9),      % Single digit integers, start at 2 to avoid 1 in denominator
        domain([N,D,LCM],1,50),             % Possible values for answer
        N1 #< D1,                           % First operand - proper fraction
        N2 #< D2,                           % Second operand - proper fraction
        D1 #\= D2,                          % Different denominators
        D #= LCM,                                   % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),           % Generate values for all variables
        lcm(D1,D2,LCM),                             % Calculate LCM
        \+ cancel(N1,D1,_,_),                       % First operand in simplest form
        \+ cancel(N2,D2,_,_),                       % Second operand in simplest form
        cancel(D1,D2,_,_),                  % Denominators are multiples of one another
        N1/D1 + N2/D2 =:= N/D.              % Arithmetic expression
```

Question Type PT5: Add a Proper and an Improper fraction of different denominators where the denominators are multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt5(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),       % Single digit integers for numerators
        domain([D1,D2], 2, 9),      % Single digit integers, start at 2 to avoid 1 in denominator
        domain([N,D],1,50),                 % Possible values for answer
        N1 #< D1,                           % First operand - proper fraction
        N2 #> D2,                           % Second operand - improper fraction
        D1 #\= D2,                          % Different denominators
        D #= LCM,                           % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),   % Generate values for all variables
        lcm(D1,D2,LCM),                     % Calculate LCM
        \+ cancel(N1,D1,_,_),               % First operand in simplest form
        \+ cancel(N2,D2,_,_),               % Second operand in simplest form
        cancel(D1,D2,_,_),                  %Denominators are multiples of one another
        N1/D1 + N2/D2 =:= N/D.              % Arithmetic expression
```

Question Type PT6: Add Two Improper fractions of different denominators where the denominators are multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt6(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),               % Single digit integers for numerators
        domain([D1,D2], 2, 9),              % Single digit integers, start at 2
                                            % to avoid 1 in denominator
        domain([N,D],1,50),                 % Possible values for answer
        N1 #> D1,                           % First operand - improper fraction
        N2 #> D2,                           % Second operand - improper fraction
        D1 #\= D2,                          % Different denominators
        D #= LCM,                           % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),   % Generate values for all variables
        lcm(D1,D2,LCM),                     % Calculate LCM
        \+ cancel(N1,D1,_,_),               % First operand in simplest form
         \+ cancel(N2,D2,_,_),              % Second operand in simplest form
        cancel(D1,D2,_,_),                  % Denominators are multiples of one another
        N1/D1 + N2/D2 =:= N/D.              % Arithmetic expression
```

Question Type PT7: Add two proper fractions of different denominators where the denominators are not multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt7(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),               % Single digit integers for numerators
        domain([D1,D2], 2, 9),      % Single digit integers, start at 2 to avoid 1 in denominator
        domain([N,D,LCM],1,50),             % Possible values for answer
        N1 #< D1,                           % First operand - proper fraction
        N2 #< D2,                           % Second operand - proper fraction
        D1 #\= D2,                          % Different denominators
        D #= LCM,                           % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),   % Generate values for all variables
        lcm(D1,D2,LCM),                             % Calculate LCM
        \+ cancel(N1,D1,_,_),               % First operand in simplest form
        \+ cancel(N2,D2,_,_),               % Second operand in simplest form
        \+ cancel(D1,D2,_,_),               % Denominators are not multiples of one another
        N1/D1 + N2/D2 =:= N/D.              % Arithmetic expression
```

Question Type PT8: Add a proper fraction and an improper fraction of different where the denominators are not multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt8(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),              % Single digit integers for numerators
        domain([D1,D2], 2, 9),     % Single digit integers, start at 2 to avoid 1 in denominator
        domain([N,D,LCM],1,50),            % Possible values for answer
        N1 #< D1,                          % First operand - proper fraction
        N2 #> D2,                          % Second operand - improper fraction
        D1 #\= D2,                         % Different denominators
        D #= LCM,                          % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),  % Generate values for all variables
        lcm(D1,D2,LCM),                    % Calculate LCM
        \+ cancel(N1,D1,_,_),              % First operand in simplest form
        \+ cancel(N2,D2,_,_),              % Second operand in simplest form
        \+ cancel(D1,D2,_,_),              % Denominators are not multiples of one another
        N1/D1 + N2/D2 =:= N/D.             % Arithmetic expression
```

Question Type PT9: Add two improper fractions of different denominators where the denominators are not multiples of one another

```
?- use_module(library(clpfd)).
% Invoke all possible answer types for N1/D1 + N2/D2 = N/D
pt9(N1,D1,N2,D2,N,D):-
        domain([N1,N2], 1,8),      % Single digit integers for numerators
        domain([D1,D2], 2, 9),     % Single digit integers, start at 2 to avoid 1 in denominator
        domain([N,D,LCM],1,50),% Possible values for answer
        N1 #> D1,                          % First operand - improper fraction
        N2 #> D2,                          % Second operand - improper fraction
        D1 #\= D2,                         % Different denominators
        D #= LCM,                          % Denominator in answer is the LCM
        labeling([], [N1,D1,N2,D2, N,D]),  % Generate values for all variables
        lcm(D1,D2,LCM),                    % Calculate LCM
        \+ cancel(N1,D1,_,_),              % First operand in simplest form
        \+ cancel(N2,D2,_,_),              % Second operand in simplest form
        \+ cancel(D1,D2,_,_),              % Denominators are not multiples of one another
        N1/D1 + N2/D2 =:= N/D.             % Arithmetic expression
```

% COMMON PREDICATES

```
cancel(N,D,X,Y) :-                         % Cancel fraction e.g. N/D into lowest form X/Y
        domain([N,D,X,Y,F], 1,99),         % F is the highest common factor
        F*X #= N,
        F*Y #= D,
        maximize(labeling([], [F,X,Y]), F),
        F \== 1.                           % To ensure cancel fraction has taken place
lcm(D1,D2, LCM) :-                         % Calculate LCM Lowest Common Multiple
    Prod is D1*D2,  gcd(D1,D2,GCD),
    LCM is Prod//GCD.
gcd(X,X,X).                                % calculate greatest common denominator
gcd(X,Y,D) :- X<Y, Y1 is Y-X, gcd(X,Y1,D).
gcd(X,Y,D) :- Y<X, gcd(Y,X,D).
```

Appendix D. Problem Classes of Fraction Additions

This appendix contains a list of problem classes characterised by possible response types for the domain of fraction additions. This was discussed in Section 3.6.2.

Problem Class PT1: Add Two Proper Fractions with Common Denominators

PT1_RT1: PT1 where response type is a proper fraction in simplest form
PT1_RT2: PT1 where response type is a whole number 1

PT1_RT3: PT1 where response type is a proper fraction not in simplest form
PT1_RT4: PT1 where response type is an improper fraction in simplest form

PT1_RT5: PT1 where response type is an improper fraction not in simplest form

Problem Class PT2: Add Two Improper Fractions with Common Denominators

PT2_RT4: PT2 where response type is an improper fraction in simplest form
PT2_RT5: PT2 where response type is an improper fraction not in simplest form

PT2_RT6: PT2 where response type is a whole number greater than 1

Problem Class PT3: Add a Proper Fraction and an Improper Fraction with Common Denominators

PT3_RT4: PT3 where response type is an improper fraction in simplest form
PT3_RT5: PT3 where response type is an improper fraction not in simplest form

PT3_RT6: PT3 where response type is a whole number greater than 1

Problem Class PT4: Add Two Proper Fractions of Different Denominators which are multiples of one another

PT4_RT1: PT4 where response type is a proper fraction in simplest form

PT4_RT3: PT4 where response type is a proper fraction not in simplest form
PT4_RT4: PT4 where response type is an improper fraction in simplest form

PT4_RT5: PT4 where response type is an improper fraction not in simplest form

Problem Class PT5: Add Two Improper Fractions of Different Denominators which are multiples of one another

PT5_RT4: PT5 where response type is an improper fraction in simplest form

PT5_RT5: PT5 where response type is an improper fraction not in simplest form

Problem Class PT6: Add a Proper Fraction and an Improper Fraction of Different Denominators which are multiples of one another

PT6_RT4: PT6 where response type is an improper fraction in simplest form

PT6_RT5: PT6 where response type is an improper fraction not in simplest form

Problem Class PT7: Add Two Proper Fractions of Different Denominators which are not multiples of one another

PT7_RT1: PT7 where response type is a proper fraction in simplest form
PT7_RT4: PT7 where response type is an improper fraction in simplest form

Problem Class PT8: Add Two Improper Fractions of Different Denominators which are not multiples of one another

PT8_RT4: PT8 where response type is an improper fraction in simplest form

Problem Class PT9: Add a Proper Fraction and an Improper Fraction of Different Denominators which are not multiples of one another

PT9_RT4: PT9 where response type is an improper fraction in simplest form

Appendix E. Fixed-Item Test in Fraction Additions

This appendix contains a fixed-item test which was administered to students. This was discussed in Section 4.2.1.3.

Assessment Test Paper

Date: ____________ Time now : ___________

Please answer all questions. At the start, please state the current time. When you finish, please state the new time on the last page.

Write down your working at all times.

1. Simplify the following fractions into its lowest form

Q1. $\frac{2}{4} =$

Q2. $\frac{8}{8} =$

Q3. $\frac{10}{5} =$

Q4. $\frac{7}{3} =$

Q5. $\frac{10}{4} =$

2. Find the Least Common Denominator for each pair of numbers. If you can't, then just give a common denominator for each pair.

Q6. 3 and 4 =

Q7. 5 and 7 =

Q8. 6 and 8 =

Q9. 4 and 6 =

3. Find an equivalent fraction for each of the following. For example, $\frac{1}{2} = \frac{2}{4}$

Q10. $\frac{3}{4} =$

Q11. $\frac{7}{3} =$

4. The following questions are addition of fractions. Please answer each question and give the answer in its lowest form. For example:

$$\frac{1}{6} + \frac{1}{6} = \frac{2}{6} = \frac{1}{3} \qquad \frac{3}{4} + \frac{3}{4} = \frac{6}{4} = 1\frac{2}{4} = 1\frac{1}{2} \qquad \frac{2}{3} + \frac{1}{3} = \frac{3}{3} = 1$$

Please show your working, if any, at all times in the space provided and write your answer in the box.

	Working	**Answer**
Q12.	$\frac{1}{3} + \frac{1}{3} =$	
Q13.	$\frac{5}{7} + \frac{2}{7} =$	

	Working	**Answer**
Q14.	$\frac{4}{9}+\frac{2}{9}=$	
Q15.	$\frac{5}{7}+\frac{6}{7}=$	
Q16.	$\frac{8}{9}+\frac{4}{9}=$	
Q17.	$\frac{4}{7}+\frac{8}{7}=$	
Q18.	$\frac{5}{8}+\frac{9}{8}=$	
Q19.	$\frac{1}{3}+\frac{5}{3}=$	

	Working	Answer
Q20.	$\frac{8}{5} + \frac{6}{5} =$	
Q21.	$\frac{5}{4} + \frac{5}{4} =$	
Q22.	$\frac{3}{2} + \frac{5}{2} =$	
Q23.	$\frac{1}{2} + \frac{1}{4} =$	
Q24.	$\frac{5}{6} + \frac{1}{8} =$	
Q25.	$\frac{1}{2} + \frac{1}{6} =$	

	Working	Answer
Q26.	$\frac{3}{8}+\frac{3}{4}=$	
Q27.	$\frac{2}{3}+\frac{5}{6}=$	
Q28.	$\frac{1}{8}+\frac{7}{4}=$	
Q29.	$\frac{1}{2}+\frac{7}{6}=$	
Q30.	$\frac{5}{3}+\frac{7}{6}=$	
Q31.	$\frac{4}{3}+\frac{7}{6}=$	

	Working	**Answer**
Q32.	$\frac{1}{2}+\frac{1}{5}=$	
Q33.	$\frac{2}{3}+\frac{3}{5}=$	
Q34.	$\frac{1}{2}+\frac{7}{3}=$	
Q35.	$\frac{5}{2}+\frac{8}{5}=$	

End of Test

Please write the time now:________

Appendix F. Diagnosing Student Answers

This appendix presents the diagnosis of student answers in two modes – evaluating final answers and inspecting intermediate steps. This was discussed in Section 4.2.1.4. Table 4 shows each question corresponding to a problem class and a set of problem solving skills which are expected to be used to solve a problem of a class. The skills are labelled in the following way:

a: add equivalent fractions
b: cancel or simplify fraction into lowest form
c: make proper
d: find common multiple or lowest common multiple
e: find equivalent fraction

Table 5 shows the results of the first mode of diagnosis and Table 6 shows the results of the second mode of diagnosis. In both tables, labels A to L correspond to each of the twelve students who undertook the test while labels Q1 to Q35 correspond to the test of thirty-five questions which is given in Appendix E. The following legend is used:

1	correct
0	incorrect
w	incorrect
na	not attempted
cbnw	correct but no working given
cbmsw	correct but missed some working
cs	possible careless slip
sc	possible copying
pc	partially correct
mq	possible misunderstanding of question
mr1	added denominators for the resultant denominator (for common denominator problems)
mr2	for non-common denominator problems, added numerators for resultant numerator and added denominators for resultant denominator
mr3	add numerators but subtract denominators (for non-common denominator problems)
mr4	added numerators and multiplied denominators (for non-common denominator problems)

Question	Problem Class	Skills likely to be applied
Q1		b
Q2		b
Q3		b
Q4		c
Q5		b,c
Q6		d
Q7		d
Q8		d
Q9		d
Q10		e
Q11		e
Q12	PT1_RT1	a
Q13	PT1_RT2	a,b
Q14	PT1_RT3	a,b
Q15	PT1_RT4	a,c
Q16	PT1_RT5	a,b,c
Q17	PT2_RT4	a,c
Q18	PT2_RT5	a,b,c
Q19	PT2_RT6	a,b
Q20	PT3_RT4	a,c
Q21	PT3_RT5	a,b,c
Q22	PT3_RT6	a,b
Q23	PT4_RT1	a,d,e
Q24	PT4_RT1	a,d,e
Q25	PT4_RT3	a,b,d,e
Q26	PT4_RT4	a,c,d,e
Q27	PT4_RT5	a,b,c,d,e
Q28	PT5_RT4	a,c,d,e
Q29	PT5_RT5	a,b,c,d,e
Q30	PT6_RT4	a,c,d,e
Q31	PT6_RT5	a,b,c,d,e
Q32	PT7_RT1	a,d,e
Q33	PT7_RT4	a,c,d,e
Q34	PT8_RT4	a,c,d,e
Q35	PT9_RT4	a,c,d,e

Table 4. Test Questions categorised by Problem Class and Skills

	A	B	C	D	E	F	G	H	I	J	K	L
Q1	1	1	1	1	1	1	1	1	1	1	1	1
Q2	1	0	1	1	1	1	1	1	1	1	1	1
Q3	0	0	1	1	1	1	1	0	1	1	1	1
Q4	0	0	0	1	0	1	1	1	1	1	1	1
Q5	0	0	0	1	0	1	1	1	1	1	1	1
Q6	0	0	1	0	1	1	1	1	0	1	1	1
Q7	0	0	1	0	1	1	1	1	0	1	1	1
Q8	0	0	1	0	0	1	1	1	0	1	1	1
Q9	0	0	0	0	0	1	1	1	0	1	1	1
Q10	1	0	0	0	0	1	0	1	1	1	1	1
Q11	0	0	0	0	0	0	0	1	0	1	1	1
Q12	0	0	1	1	1	1	0	1	1	1	1	1
Q13	0	0	1	1	1	1	1	1	1	1	1	1
Q14	0	0	0	1	1	1	1	1	1	1	1	1
Q15	0	0	0	1	1	1	1	1	1	1	1	1
Q16	0	0	0	1	1	1	1	1	1	1	1	1
Q17	0	0	0	1	1	1	1	1	0	1	1	1
Q18	0	0	0	0	1	1	1	1	1	1	1	1
Q19	0	0	0	1	1	1	0	1	1	1	1	1
Q20	0	0	0	1	1	1	1	1	0	1	1	1
Q21	0	0	0	1	1	1	0	1	1	0	1	1
Q22	0	0	0	1	1	1	1	1	1	1	1	1
Q23	0	0	0	1	1	1	1	1	1	1	1	1
Q24	0	0	0	0	0	1	1	1	1	1	1	1
Q25	0	0	0	0	0	1	1	1	1	1	1	1
Q26	0	0	0	0	0	1	1	1	1	1	1	1
Q27	0	0	0	0	0	1	1	1	1	1	1	1
Q28	0	0	0	0	0	1	1	1	1	1	1	1
Q29	0	0	0	0	0	0	1	1	1	1	1	1
Q30	0	0	0	0	0	1	1	1	1	1	1	1
Q31	0	0	0	0	0	0	1	1	1	1	1	1
Q32	0	0	0	0	0	0	1	1	1	1	1	1
Q33	0	0	0	0	0	0	1	1	1	0	1	1
Q34	0	0	0	0	0	0	1	1	1	1	1	1
Q35	0	0	0	0	0	0	1	1	1	1	1	1

Table 5. Evaluating Final Answers only

	A	B	C	D	E	F	G	H	I	J	K	L
Q1	1	1	1	1	1	1	1	1	1	1	1	1
Q2	1	w	1	1	1	1	1	1	1	1	1	1
Q3	na	w	1	1	1	1	1	w	1	1	1	1
Q4	na	w	W	1	w	1	1	1	1	1	1	1
Q5	na	w	W	1	pc	1	1	1	1	1	1	1
Q6	w	na	1	na	1	1	1	1	w	1	1	1
Q7	w	na	1	na	1	1	1	1	w	1	1	1
Q8	w	na	1	na	w	1	1	1	w	1	1	1
Q9	w	na	W	na	w	1	1	1	w	1	1	1
Q10	1	na	W	mq	w	1	w	1	1	1	1	1
Q11	na	na	W	mq	w	w	w	1	w	1	1	1
Q12	mr1	na	1	1	1	1	w	1	1	1	1	1
Q13	mr1	na	1	1	1	1	1	1	1	1	1	1
Q14	mr1	na	W	1	1	1	1	1	1	1	1	1
Q15	mr1	na	W	1	1	1	cbnw	1	1	1	1	1
Q16	mr1	na	W	1	1	1	cbnw	1	cbmsw	1	1	1
Q17	mr1	na	W	1	1	1	cbnw	1	pc	1	1	1
Q18	na	na	W	cs	1	1	cbmsw	1	cbmsw	1	1	1
Q19	na	na	W	1	1	1	pc	1	1	1	1	1
Q20	na	na	W	1	1	1	1	1	pc	1	1	1
Q21	na	na	W	1	1	1	na	1	1	w	1	1
Q22	na	na	W	1	1	1	1	1	1	1	1	1
Q23	na	na	W	cbnw	cbnw	1	1	1	1	1	1	1
Q24	na	na	W	w	w	1	1	1	cbnw/sc	1	1	1
Q25	na	na	W	w	w	1	1	1	cbnw/sc	1	1	1
Q26	na	na	W	mr2	w	1	1	1	cbnw/sc	1	1	1
Q27	na	na	W	mr2	w	1	1	1	cbnw/sc	1	1	1
Q28	na	na	W	mr2	w	1	1	1	cbnw/sc	1	1	1
Q29	na	na	W	mr2	mr3	cs	1	1	cbnw/sc	1	1	1
Q30	na	na	W	mr2	mr3	1	1	1	cbnw/sc	1	1	1
Q31	na	na	W	mr2	mr3	na	1	1	cbnw/sc	1	1	1
Q32	na	na	W	mr2	mr4	na	1	1	cbnw/sc	1	1	1
Q33	na	na	W	mr2	mr4	na	1	1	cbnw/sc	cs	1	1
Q34	Na	na	W	mr2	mr4	na	1	1	cbnw/sc	1	1	1
Q35	Na	na	W	mr2	mr4	na	1	1	cbnw/sc	1	1	1

Table 6. Inspecting Solution Paths and Final Answers

Appendix G. Simulated Students

This appendix presents two pieces of information. Firstly, a list of instances of simulated students of *Sam1, Sam2, Sam3, Sam4* and *Sam5* types, as discussed in Section 6.3, is presented in Table 7 and Table 8. The last three columns of the tables list the names of the output files generated from running the steps of the evaluation - “generate logfiles” and “running the assessor”, as described in Sections 6.4 and 6.5 respectively.

The second piece of information is a list of the instantiations of the simulated students – see Figure 36. The instantiations are represented as a Prolog predicate, *simStudents(X,Y)*, where *X* is the name of the simulated student and *Y* is a three-phase structure, *Prepare-Add-Tidy*, where each phase contains one or more mastered skill or malrule.

Type	**Student**	**Prepare Phase**	**Process Phase**	**Tidy Phase**	**Logfiles**	**Run XP**	**Run ST**
Sam1	sam1a	makeVulgar, makeCommon	checkAndAdd	cancel, makeProper, makeWhole	log1a	xp_1a	st_1a
(knows all	sam1b	makeCommon, makeVulgar	checkAndAdd	cancel, makeProper, makeWhole	log1b	xp _1b	st _1b
the skills)	sam1c	makeVulgar, makeCommon	checkAndAdd	makeProper, makeWhole,cancel	log1c	xp _1c	st _1c
	sam1d	makeVulgar, makeCommon	checkAndAdd	makeWhole,cancel, makeProper	log1d	xp _1d	st _1d
	sam1e	makeVulgar, makeCommon	checkAndAdd	cancel, makeWhole makeProper	log1e	xp _1e	st _1e
	sam1f	makeCommon, makeVulgar	checkAndAdd	makeProper, makeWhole,cancel	log1f	xp _1f	st _1f
	sam1g	makeCommon, makeVulgar	checkAndAdd	makeWhole,cancel, makeProper	log1g	xp _1g	st _1g
	sam1h	makeCommon, makeVulgar	checkAndAdd	cancel, makeWhole makeProper	log1h	xp _1h	st _1h
Sam2	sam2a	makeVulgar	checkAndAdd	cancel, makeProper, makeWhole	log2a	xp _2a	st _2a
(gaps in	sam2b	makeCommon	checkAndAdd	cancel, makeProper, makeWhole	log2b	xp _2b	st _2b
knowledge)	sam2c	makeVulgar, makeCommon	checkAndAdd	makeProper, makeWhole	log2c	xp _2c	st _2c
	sam2d	makeVulgar, makeCommon	checkAndAdd	cancel, makeWhole	log2d	xp _2d	st _2d
	sam2e	makeVulgar, makeCommon	checkAndAdd	cancel, makeProper	log2e	xp _2e	st _2e
	sam2f	makeCommon	checkAndAdd	-	log2f	xp _2f	st _2f

Table 7. Simulated Students with overlay knowledge

Type	Student	Prepare Phase	Process Phase	Tidy Phase	Logfiles	Run XP	Run ST
Sam3	sam3a	makeVulgar, makeCommon	checkAndAdd	malCancel, makeProper, makeWhole	log3a	xp_ 3a	st _ 3a
(malrules)	sam3b	makeVulgar, makeCommon	malAdd1	cancel, makeProper, makeWhole	log3b	xp _ 3b	st _ 3b
	sam3c	MakeVulgar	malAdd2	cancel, makeProper, makeWhole	log3c	xp _3c	st _3c
	sam3d	MakeVulgar	malAdd3	cancel, makeProper, makeWhole	log3d	xp _ 3d	st _ 3d
	sam3e	makeVulgar	malAdd1	malCancel, makeProper, makeWhole	log3e	xp _ 3e	st _ 3e
Sam4	sam4a	makeCommon	checkAndAdd	-	log4a	xp _ 4a	st _ 4a
(guesses)	sam4b	makeCommon	checkAndAdd	-	log4b	xp _ 4b	st _ 4b
	sam4c	makeCommon	checkAndAdd	-	log4c	xp _ 4c	st _ 4c
	sam4d	makeVulgar	checkAndAdd	cancel, makeProper, makeWhole	log4d	xp _4d	st _4d
	sam4e	makeVulgar, makeCommon	checkAndAdd	makeProper, makeWhole	log4e	xp _4e	st _4e
Sam5	sam5a	makeVulgar, makeCommon	checkAndAdd	cancel, makeProper, makeWhole	log5a	xp _ 5a	st _ 5a
(slips)	sam5b	makeVulgar, makeCommon	checkAndAdd	cancel, makeProper, makeWhole	log5b	xp _ 5b	st _ 5b
	sam5c	makeVulgar, makeCommon	checkAndAdd	makeProper, makeWhole,cancel	log5c	xp _5c	st _5c
	sam5d	MakeCommon	checkAndAdd	cancel, makeProper, makeWhole	log5d	xp _5d	st _5d

Table 8. Simulated Students with noisy data

```
/* simStudents(X,Y) where X is the simulated student and Y is the three-phase structure of student X */
% Sam1 - students who knows all the relevant skills - differing orders
simStudents(sam1a, [[sam1a-[[makeVulgar, makeCommon],[checkAndAdd],[cancel, makeProper, makeWhole]]]]).
simStudents(sam1b, [[sam1b-[[makeCommon, makeVulgar],[checkAndAdd],[cancel, makeProper, makeWhole]]]]).
simStudents(sam1c, [[sam1c-[[makeVulgar, makeCommon],[checkAndAdd],[makeProper, makeWhole, cancel]]]]).
simStudents(sam1d, [[sam1d-[[makeVulgar, makeCommon],[checkAndAdd],[makeWhole, cancel, makeProper]]]]).
simStudents(sam1e, [[sam1e-[[makeVulgar, makeCommon],[checkAndAdd],[cancel, makeWhole, makeProper]]]]).
simStudents(sam1f, [[sam1f-[[makeCommon,makeVulgar],[checkAndAdd],[makeProper, makeWhole, cancel]]]]).
simStudents(sam1g, [[sam1g-[[makeCommon,makeVulgar],[checkAndAdd],[makeWhole, cancel, makeProper]]]]).
simStudents(sam1h, [[sam1h-[[makeCommon,makeVulgar],[checkAndAdd],[cancel, makeWhole, makeProper]]]]).
% Sam2 - students with gaps in knowledge
simStudents(sam2a, [[sam2a-[[makeVulgar],[checkAndAdd],[cancel,makeProper,makeWhole]]]]).
simStudents(sam2b, [[sam2b-[[makeCommon],[checkAndAdd],[cancel, makeProper, makeWhole]]]]).
simStudents(sam2c, [[sam2c-[[makeVulgar, makeCommon],[checkAndAdd],[makeProper, makeWhole]]]]).
simStudents(sam2d, [[sam2d-[[makeVulgar, makeCommon],[checkAndAdd],[cancel, makeWhole]]]]).
simStudents(sam2e, [[sam2e-[[makeVulgar, makeCommon],[checkAndAdd],[cancel, makeProper]]]]).
simStudents(sam2f, [[sam2f-[[makeCommon],[checkAndAdd],[]]]]).
% Sam3 - students with malrules
simStudents(sam3a,[[sam3a-[[makeVulgar, makeCommon],[checkAndAdd],[malCancel, makeProper, makeWhole]]]]).
simStudents(sam3b,[[sam3b-[[makeVulgar, makeCommon],[malAdd1],[cancel, makeProper, makeWhole]]]]).
simStudents(sam3c,[[sam3c-[[makeVulgar],[malAdd2],[cancel, makeProper, makeWhole]]]]).
simStudents(sam3d,[[sam3d-[[makeVulgar],[malAdd3],[cancel, makeProper, makeWhole]]]]).
simStudents(sam3e,[[sam3e-[[makeVulgar],[malAdd1],[malCancel, makeProper, makeWhole]]]]).
% Sam4 - students with lucky guesses
simStudents(sam4a,[[sam4a-[[makeCommon],[checkAndAdd],[]]]]).
simStudents(sam4b,[[sam4b-[[makeCommon],[checkAndAdd],[]]]]).
simStudents(sam4c,[[sam4c-[[makeCommon],[checkAndAdd],[]]]]).
simStudents(sam4d,[[sam4d-[[ makeVulgar],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
simStudents(sam4e,[[sam4e-[[ makeVulgar,makeCommon],[checkAndAdd],[makeProper, makeWhole]]]]).
% Sam5 - students with careless slips
simStudents(sam5a,[[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
simStudents(sam5b,[[sam5b-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
simStudents(sam5c,[[sam5c-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
simStudents(sam5d,[[sam5d-[[makeCommon],[checkAndAdd],[cancel,makeProper, makeWhole]]]]).
```

Figure 36. Prolog Instantiations of Simulated Students

Appendix H. Set of Fraction Additions Problems

This appendix contains a database of 68 problems in fraction additions used during the evaluation of the XP assessor, as discussed in Sections 6.4 and 6.5. Some of the questions are drawn from the DSA experiment (Section 4.2.1).

1 skill problems:

question(1,[c], q_c_1, fr(1/3,1/3), fr(2/3)).
question(1,[c], q_c_2, fr(3/5,1/5), fr(4/5)).

2 skill problems:
question(2,[b,c], q_bc_1, fr(1/2,1/5), fr(7/10)).
question(2,[b,c], q_bc_2, fr(1/3,1/5), fr(8/15)).
question(2,[b,c], q_bc_3, fr(5/8,1/6), fr(19/24)).
question(2,[c,d], q_cd_1, fr(4/9,2/9), fr(2/3)).
question(2,[c,d], q_cd_2, fr(12/64,4/64), fr(1/4)).
question(2,[c,d], q_cd_3, fr(9/24,3/24), fr(1/2)).
question(2,[c,d], q_cd_4, fr(9/16,3/16), fr(3/4)).
question(2,[c,d], q_cd_5, fr(1/6,1/6), fr(1/3)).
question(2,[c,d], q_cd_6, fr(3/8,1/8), fr(1/2)).
question(2,[c,e], q_ce_1, fr(5/7,6/7), fr(1:4/7)).
question(2,[c,e], q_ce_2, fr(4/7,8/7), fr(1:5/7)).
question(2,[c,e], q_ce_3, fr(8/5,6/5), fr(2:4/5)).
question(2,[c,e], q_ce_4, fr(2/3,2/3), fr(1:1/3)).
question(2,[c,e], q_ce_5, fr(5/6,2/6), fr(1:1/6)).
question(2,[c,e], q_ce_6, fr(7/5,2/5), fr(1:4/5)).

3 skill problems:
question(3,[a,c,e], q_ace_1, fr(1:1/5,2/5), fr(1:3/5)).
question(3,[a,c,e], q_ace_2, fr(2:3/7,2/7), fr(2:5/7)).
question(3,[a,c,e], q_ace_3, fr(1:1/5,3/5), fr(1:4/5)).
question(3,[b,c,d], q_bcd_1, fr(1/2,1/4), fr(3/4)).
question(3,[b,c,d], q_bcd_2, fr(5/6,1/8), fr(23/24)).
question(3,[b,c,d], q_bcd_3, fr(1/2,1/6), fr(2/3)).
question(3,[b,c,d], q_bcd_4, fr(1/12,1/6), fr(1/4)).
question(3,[b,c,d], q_bcd_5, fr(2/15,1/5), fr(1/3)).

```
question(3,[b,c,e], q_bce_1, fr(2/3,3/5), fr(1:4/15)).
question(3,[b,c,e], q_bce_2, fr(1/2,7/3), fr(2:5/6)).
question(3,[b,c,e], q_bce_3, fr(5/2,8/5), fr(4:1/10)).
question(3,[b,c,e], q_bce_4, fr(4/5,3/4), fr(1:11/20)).
question(3,[b,c,e], q_bce_5, fr(5/6,1/3), fr(1:1/6)).
question(3,[b,c,e], q_bce_6, fr(6/7,3/8), fr(1:13/56)).
question(3,[b,c,e], q_bce_7, fr(6/7,5/8), fr(1:27/56)).
question(3,[b,c,e], q_bce_8, fr(5/8,4/5), fr(1:17/40)).
question(3,[b,c,e], q_bce_9, fr(5/8,5/6), fr(1:11/24)).
question(3,[b,c,e], q_bce_10, fr(3/8,5/6), fr(1:5/24)).
question(3,[b,c,e], q_bce_11, fr(8/9,3/5), fr(1:22/45)).
question(3,[c,d,e], q_cde_1, fr(5/8,9/8), fr(1:3/4)).
question(3,[c,d,e], q_cde_2, fr(8/9,4/9), fr(1:1/3)).
question(3,[c,d,e], q_cde_3, fr(5/4,5/4), fr(2:1/2)).
question(3,[c,d,f], q_cdf_1, fr(5/7,2/7), fr(1)).
question(3,[c,d,f], q_cdf_2, fr(4/5,1/5), fr(1)).
question(3,[c,d,f], q_cdf_3, fr(1/2,1/2), fr(1)).
question(3,[c,d,f], q_cdf_4, fr(4/5,1/5), fr(1)).
question(3,[c,d,f], q_cdf_5, fr(1/2,1/2), fr(1)).
```

4 skill problems:

```
question(4,[a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), fr(3:4/15)).
question(4,[a,b,c,e], q_abce_2, fr(2:1/2,7/3), fr(4:5/6)).
question(4,[a,b,c,e], q_abce_3, fr(1/4,2:1/8), fr(2:3/8)).
question(4,[a,b,c,e], q_abce_4, fr(1:3/8,1/2), fr(1:7/8)).
question(4,[a,b,c,e], q_abce_5, fr(1/5,1:5/7), fr(1:32/35)).
question(4,[a,c,e,f], q_acef_1, fr(1:1/3,2/3), fr(2)).
question(4,[a,c,e,f], q_acef_2, fr(1:1/5,4/5), fr(2)).
question(4,[c,d,e,f], q_cdef_1, fr(1/3,5/3), fr(2)).
question(4,[c,d,e,f], q_cdef_2, fr(3/2,5/2), fr(4)).
question(4,[b,c,d,e], q_bcde_1, fr(3/8,3/4), fr(1:1/8)).
question(4,[b,c,d,e], q_bcde_2, fr(2/3,5/6), fr(1:1/2)).
question(4,[b,c,d,e], q_bcde_3, fr(1/8,7/4), fr(1:7/8)).
question(4,[b,c,d,e], q_bcde_4, fr(1/2,7/6), fr(1:2/3)).
question(4,[b,c,d,e], q_bcde_5, fr(5/3,7/6), fr(2:5/6)).
question(4,[b,c,d,e], q_bcde_6, fr(4/3,7/6), fr(2:1/2)).
question(4,[b,c,d,e], q_bcde_7, fr(3/8,5/6), fr(1:5/24)).
question(4,[b,c,d,e], q_bcde_8, fr(4/5,3/10), fr(1:1/10)).
question(4,[b,c,d,e], q_bcde_9, fr(7/8,3/4), fr(1:5/8)).
question(4,[b,c,d,f], q_bcdf_1, fr(3/6,2/4), fr(1)).
question(4,[b,c,d,f], q_bcdf_2, fr(5/10,4/8), fr(1)).
question(4,[a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), fr(2:1/2)).
question(4,[a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), fr(3:1/3)).
```

5 skill problems:

```
question(5,[a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), fr(2:1/6)).
question(5,[a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), fr(2:1/3)).
```

Appendix I. Generated Logfiles

This appendix contains a selected list of logfiles, generated for different student types. This was described in Section 6.4. The logfiles use the following legend to represent the different skills and malrules:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole
mg	malCancel
mh	malAdd1
mi	malAdd2
mj	malAdd3

Logfile: log1a

simStudents([[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]).

```
sam(sam1a,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam1a,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam1a,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam1a,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam1a,[b,c,d],q_bc_3,fr(5/8,1/6),fr(19/24),ok).
sam(sam1a,[c,d],q_cd_1,fr(4/9,2/9),fr(2/3),ok).
sam(sam1a,[c,d],q_cd_2,fr(12/64,4/64),fr(1/4),ok).
sam(sam1a,[c,d],q_cd_3,fr(9/24,3/24),fr(1/2),ok).
sam(sam1a,[c,d],q_cd_4,fr(9/16,3/16),fr(3/4),ok).
sam(sam1a,[c,d],q_cd_5,fr(1/6,1/6),fr(1/3),ok).
sam(sam1a,[c,d],q_cd_6,fr(3/8,1/8),fr(1/2),ok).
sam(sam1a,[c,e],q_ce_1,fr(5/7,6/7),fr(1:4/7),ok).
sam(sam1a,[c,e],q_ce_2,fr(4/7,8/7),fr(1:5/7),ok).
sam(sam1a,[c,e],q_ce_3,fr(8/5,6/5),fr(2:4/5),ok).
sam(sam1a,[c,e],q_ce_4,fr(2/3,2/3),fr(1:1/3),ok).
sam(sam1a,[c,e],q_ce_5,fr(5/6,2/6),fr(1:1/6),ok).
sam(sam1a,[c,e],q_ce_6,fr(7/5,2/5),fr(1:4/5),ok).
sam(sam1a,[a,c,e],q_ace_1,fr(1:1/5,2/5),fr(1:3/5),ok).
sam(sam1a,[a,c,e],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam1a,[a,c,e],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam1a,[b,c,d],q_bcd_1,fr(1/2,1/4),fr(3/4),ok).
sam(sam1a,[b,c,d],q_bcd_2,fr(5/6,1/8),fr(23/24),ok).
sam(sam1a,[b,c,d],q_bcd_3,fr(1/2,1/6),fr(2/3),ok).
sam(sam1a,[b,c,d],q_bcd_4,fr(1/12,1/6),fr(1/4),ok).
sam(sam1a,[b,c,d],q_bcd_5,fr(2/15,1/5),fr(1/3),ok).
sam(sam1a,[b,c,e],q_bce_1,fr(2/3,3/5),fr(1:4/15),ok).
sam(sam1a,[b,c,e],q_bce_2,fr(1/2,7/3),fr(2:5/6),ok).
sam(sam1a,[b,c,e],q_bce_3,fr(5/2,8/5),fr(4:1/10),ok).
sam(sam1a,[b,c,e],q_bce_4,fr(4/5,3/4),fr(1:11/20),ok).
sam(sam1a,[b,c,d,e],q_bce_5,fr(5/6,1/3),fr(1:1/6),ok).
sam(sam1a,[b,c,e],q_bce_6,fr(6/7,3/8),fr(1:13/56),ok).
sam(sam1a,[b,c,e],q_bce_7,fr(6/7,5/8),fr(1:27/56),ok).
sam(sam1a,[b,c,e],q_bce_8,fr(5/8,4/5),fr(1:17/40),ok).
sam(sam1a,[b,c,d,e],q_bce_9,fr(5/8,5/6),fr(1:11/24),ok).
sam(sam1a,[b,c,d,e],q_bce_10,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam1a,[b,c,e],q_bce_11,fr(8/9,3/5),fr(1:22/45),ok).
sam(sam1a,[c,d,e],q_cde_1,fr(5/8,9/8),fr(1:3/4),ok).
sam(sam1a,[c,d,e],q_cde_2,fr(8/9,4/9),fr(1:1/3),ok).
sam(sam1a,[c,d,e],q_cde_3,fr(5/4,5/4),fr(2:1/2),ok).
sam(sam1a,[c,d,f],q_cdf_1,fr(5/7,2/7),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_2,fr(4/5,1/5),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_3,fr(1/2,1/2),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_4,fr(4/5,1/5),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_5,fr(1/2,1/2),fr(1),ok).
sam(sam1a,[a,b,c,e],q_abce_1,fr(1:2/3,1:3/5),fr(3:4/15),ok).
sam(sam1a,[a,b,c,e],q_abce_2,fr(2:1/2,7/3),fr(4:5/6),ok).
sam(sam1a,[a,b,c,d,e],q_abce_3,fr(1/4,2:1/8),fr(2:3/8),ok).
sam(sam1a,[a,b,c,d,e],q_abce_4,fr(1:3/8,1/2),fr(1:7/8),ok).
sam(sam1a,[a,b,c,e],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam1a,[a,c,d,e,f],q_acef_1,fr(1:1/3,2/3),fr(2),ok).
sam(sam1a,[a,c,d,e,f],q_acef_2,fr(1:1/5,4/5),fr(2),ok).
sam(sam1a,[c,d,e,f],q_cdef_1,fr(1/3,5/3),fr(2),ok).
sam(sam1a,[c,d,e,f],q_cdef_2,fr(3/2,5/2),fr(4),ok).
sam(sam1a,[b,c,d,e],q_bcde_1,fr(3/8,3/4),fr(1:1/8),ok).
sam(sam1a,[b,c,d,e],q_bcde_2,fr(2/3,5/6),fr(1:1/2),ok).
sam(sam1a,[b,c,d,e],q_bcde_3,fr(1/8,7/4),fr(1:7/8),ok).
sam(sam1a,[b,c,d,e],q_bcde_4,fr(1/2,7/6),fr(1:2/3),ok).
sam(sam1a,[b,c,d,e],q_bcde_5,fr(5/3,7/6),fr(2:5/6),ok).
sam(sam1a,[b,c,d,e],q_bcde_6,fr(4/3,7/6),fr(2:1/2),ok).
sam(sam1a,[b,c,d,e],q_bcde_7,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam1a,[b,c,d,e],q_bcde_8,fr(4/5,3/10),fr(1:1/10),ok).
sam(sam1a,[b,c,d,e],q_bcde_9,fr(7/8,3/4),fr(1:5/8),ok).
sam(sam1a,[b,c,d,f],q_bcdf_1,fr(3/6,2/4),fr(1),ok).
sam(sam1a,[b,c,d,f],q_bcdf_2,fr(5/10,4/8),fr(1),ok).
sam(sam1a,[a,c,d,e],q_acde_1,fr(1:1/8,1:3/8),fr(2:1/2),ok).
sam(sam1a,[a,c,d,e],q_acde_2,fr(1:1/6,2:1/6),fr(3:1/3),ok).
sam(sam1a,[a,b,c,d,e],q_abcde_1,fr(1/3,1:5/6),fr(2:1/6),ok).
sam(sam1a,[a,b,c,d,e],q_abcde_2,fr(1/4,2:1/12),fr(2:1/3),ok).
```

Logfile: log2e

simStudents([[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]).

```
sam(sam2e,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam2e,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam2e,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam2e,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam2e,[b,c,d],q_bc_3,fr(5/8,1/6),fr(19/24),ok).
sam(sam2e,[c,d],q_cd_1,fr(4/9,2/9),fr(2/3),ok).
sam(sam2e,[c,d],q_cd_2,fr(12/64,4/64),fr(1/4),ok).
sam(sam2e,[c,d],q_cd_3,fr(9/24,3/24),fr(1/2),ok).
sam(sam2e,[c,d],q_cd_4,fr(9/16,3/16),fr(3/4),ok).
sam(sam2e,[c,d],q_cd_5,fr(1/6,1/6),fr(1/3),ok).
sam(sam2e,[c,d],q_cd_6,fr(3/8,1/8),fr(1/2),ok).
sam(sam2e,[c,e],q_ce_1,fr(5/7,6/7),fr(1:4/7),ok).
sam(sam2e,[c,e],q_ce_2,fr(4/7,8/7),fr(1:5/7),ok).
sam(sam2e,[c,e],q_ce_3,fr(8/5,6/5),fr(2:4/5),ok).
sam(sam2e,[c,e],q_ce_4,fr(2/3,2/3),fr(1:1/3),ok).
sam(sam2e,[c,e],q_ce_5,fr(5/6,2/6),fr(1:1/6),ok).
sam(sam2e,[c,e],q_ce_6,fr(7/5,2/5),fr(1:4/5),ok).
sam(sam2e,[a,c,e],q_ace_1,fr(1:1/5,2/5),fr(1:3/5),ok).
sam(sam2e,[a,c,e],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam2e,[a,c,e],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam2e,[b,c,d],q_bcd_1,fr(1/2,1/4),fr(3/4),ok).
sam(sam2e,[b,c,d],q_bcd_2,fr(5/6,1/8),fr(23/24),ok).
sam(sam2e,[b,c,d],q_bcd_3,fr(1/2,1/6),fr(2/3),ok).
sam(sam2e,[b,c,d],q_bcd_4,fr(1/12,1/6),fr(1/4),ok).
sam(sam2e,[b,c,d],q_bcd_5,fr(2/15,1/5),fr(1/3),ok).
sam(sam2e,[b,c,e],q_bce_1,fr(2/3,3/5),fr(1:4/15),ok).
sam(sam2e,[b,c,e],q_bce_2,fr(1/2,7/3),fr(2:5/6),ok).
sam(sam2e,[b,c,e],q_bce_3,fr(5/2,8/5),fr(4:1/10),ok).
sam(sam2e,[b,c,e],q_bce_4,fr(4/5,3/4),fr(1:11/20),ok).
sam(sam2e,[b,c,d,e],q_bce_5,fr(5/6,1/3),fr(1:1/6),ok).
sam(sam2e,[b,c,e],q_bce_6,fr(6/7,3/8),fr(1:13/56),ok).
sam(sam2e,[b,c,e],q_bce_7,fr(6/7,5/8),fr(1:27/56),ok).
sam(sam2e,[b,c,e],q_bce_8,fr(5/8,4/5),fr(1:17/40),ok).
sam(sam2e,[b,c,d,e],q_bce_9,fr(5/8,5/6),fr(1:11/24),ok).
sam(sam2e,[b,c,d,e],q_bce_10,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam2e,[b,c,e],q_bce_11,fr(8/9,3/5),fr(1:22/45),ok).
sam(sam2e,[c,d,e],q_cde_1,fr(5/8,9/8),fr(1:3/4),ok).
sam(sam2e,[c,d,e],q_cde_2,fr(8/9,4/9),fr(1:1/3),ok).
sam(sam2e,[c,d,e],q_cde_3,fr(5/4,5/4),fr(2:1/2),ok).
sam(sam2e,[c,d],q_cdf_1,fr(5/7,2/7),fr(1/1),no).
sam(sam2e,[c,d],q_cdf_2,fr(4/5,1/5),fr(1/1),no).
sam(sam2e,[c,d],q_cdf_3,fr(1/2,1/2),fr(1/1),no).
sam(sam2e,[c,d],q_cdf_4,fr(4/5,1/5),fr(1/1),no).
sam(sam2e,[c,d],q_cdf_5,fr(1/2,1/2),fr(1/1),no).
sam(sam2e,[a,b,c,e],q_abce_1,fr(1:2/3,1:3/5),fr(3:4/15),ok).
sam(sam2e,[a,b,c,e],q_abce_2,fr(2:1/2,7/3),fr(4:5/6),ok).
sam(sam2e,[a,b,c,d,e],q_abce_3,fr(1/4,2:1/8),fr(2:3/8),ok).
sam(sam2e,[a,b,c,d,e],q_abce_4,fr(1:3/8,1/2),fr(1:7/8),ok).
sam(sam2e,[a,b,c,e],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam2e,[a,c,d,e],q_acef_1,fr(1:1/3,2/3),fr(2:0/1),no).
sam(sam2e,[a,c,d,e],q_acef_2,fr(1:1/5,4/5),fr(2:0/1),no).
sam(sam2e,[c,d,e],q_cdef_1,fr(1/3,5/3),fr(2:0/1),no).
sam(sam2e,[c,d,e],q_cdef_2,fr(3/2,5/2),fr(4:0/1),no).
sam(sam2e,[b,c,d,e],q_bcde_1,fr(3/8,3/4),fr(1:1/8),ok).
sam(sam2e,[b,c,d,e],q_bcde_2,fr(2/3,5/6),fr(1:1/2),ok).
sam(sam2e,[b,c,d,e],q_bcde_3,fr(1/8,7/4),fr(1:7/8),ok).
sam(sam2e,[b,c,d,e],q_bcde_4,fr(1/2,7/6),fr(1:2/3),ok).
sam(sam2e,[b,c,d,e],q_bcde_5,fr(5/3,7/6),fr(2:5/6),ok).
sam(sam2e,[b,c,d,e],q_bcde_6,fr(4/3,7/6),fr(2:1/2),ok).
sam(sam2e,[b,c,d,e],q_bcde_7,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam2e,[b,c,d,e],q_bcde_8,fr(4/5,3/10),fr(1:1/10),ok).
sam(sam2e,[b,c,d,e],q_bcde_9,fr(7/8,3/4),fr(1:5/8),ok).
sam(sam2e,[b,c,d],q_bcdf_1,fr(3/6,2/4),fr(1/1),no).
sam(sam2e,[b,c,d],q_bcdf_2,fr(5/10,4/8),fr(1/1),no).
sam(sam2e,[a,c,d,e],q_acde_1,fr(1:1/8,1:3/8),fr(2:1/2),ok).
sam(sam2e,[a,c,d,e],q_acde_2,fr(1:1/6,2:1/6),fr(3:1/3),ok).
sam(sam2e,[a,b,c,d,e],q_abcde_1,fr(1/3,1:5/6),fr(2:1/6),ok).
sam(sam2e,[a,b,c,d,e],q_abcde_2,fr(1/4,2:1/12),fr(2:1/3),ok).
```

Logfile: log2f

simStudents([[sam2f-[[makeCommon],[checkAndAdd],[]]]]).

```
sam(sam2f,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam2f,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam2f,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam2f,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam2f,[b,c],q_bc_3,fr(5/8,1/6),fr(38/48),no).
sam(sam2f,[c],q_cd_1,fr(4/9,2/9),fr(6/9),no).
sam(sam2f,[c],q_cd_2,fr(12/64,4/64),fr(16/64),no).
sam(sam2f,[c],q_cd_3,fr(9/24,3/24),fr(12/24),no).
sam(sam2f,[c],q_cd_4,fr(9/16,3/16),fr(12/16),no).
sam(sam2f,[c],q_cd_5,fr(1/6,1/6),fr(2/6),no).
sam(sam2f,[c],q_cd_6,fr(3/8,1/8),fr(4/8),no).
sam(sam2f,[c],q_ce_1,fr(5/7,6/7),fr(11/7),no).
sam(sam2f,[c],q_ce_2,fr(4/7,8/7),fr(12/7),no).
sam(sam2f,[c],q_ce_3,fr(8/5,6/5),fr(14/5),no).
sam(sam2f,[c],q_ce_4,fr(2/3,2/3),fr(4/3),no).
sam(sam2f,[c],q_ce_5,fr(5/6,2/6),fr(7/6),no).
sam(sam2f,[c],q_ce_6,fr(7/5,2/5),fr(9/5),no).
sam(sam2f,[c],q_ace_1,fr(1:1/5,2/5),fr(1:3/5),ok).
sam(sam2f,[c],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam2f,[c],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam2f,[b,c],q_bcd_1,fr(1/2,1/4),fr(6/8),no).
sam(sam2f,[b,c],q_bcd_2,fr(5/6,1/8),fr(46/48),no).
sam(sam2f,[b,c],q_bcd_3,fr(1/2,1/6),fr(8/12),no).
sam(sam2f,[b,c],q_bcd_4,fr(1/12,1/6),fr(18/72),no).
sam(sam2f,[b,c],q_bcd_5,fr(2/15,1/5),fr(25/75),no).
sam(sam2f,[b,c],q_bce_1,fr(2/3,3/5),fr(19/15),no).
sam(sam2f,[b,c],q_bce_2,fr(1/2,7/3),fr(17/6),no).
sam(sam2f,[b,c],q_bce_3,fr(5/2,8/5),fr(41/10),no).
sam(sam2f,[b,c],q_bce_4,fr(4/5,3/4),fr(31/20),no).
sam(sam2f,[b,c],q_bce_5,fr(5/6,1/3),fr(21/18),no).
sam(sam2f,[b,c],q_bce_6,fr(6/7,3/8),fr(69/56),no).
sam(sam2f,[b,c],q_bce_7,fr(6/7,5/8),fr(83/56),no).
sam(sam2f,[b,c],q_bce_8,fr(5/8,4/5),fr(57/40),no).
sam(sam2f,[b,c],q_bce_9,fr(5/8,5/6),fr(70/48),no).
sam(sam2f,[b,c],q_bce_10,fr(3/8,5/6),fr(58/48),no).
sam(sam2f,[b,c],q_bce_11,fr(8/9,3/5),fr(67/45),no).
sam(sam2f,[c],q_cde_1,fr(5/8,9/8),fr(14/8),no).
sam(sam2f,[c],q_cde_2,fr(8/9,4/9),fr(12/9),no).
sam(sam2f,[c],q_cde_3,fr(5/4,5/4),fr(10/4),no).
sam(sam2f,[c],q_cdf_1,fr(5/7,2/7),fr(7/7),no).
sam(sam2f,[c],q_cdf_2,fr(4/5,1/5),fr(5/5),no).
sam(sam2f,[c],q_cdf_3,fr(1/2,1/2),fr(2/2),no).
sam(sam2f,[c],q_cdf_4,fr(4/5,1/5),fr(5/5),no).
sam(sam2f,[c],q_cdf_5,fr(1/2,1/2),fr(2/2),no).
sam(sam2f,[b,c],q_abce_1,fr(1:2/3,1:3/5),fr(2:19/15),no).
sam(sam2f,[b,c],q_abce_2,fr(2:1/2,7/3),fr(2:17/6),no).
sam(sam2f,[b,c],q_abce_3,fr(1/4,2:1/8),fr(2:12/32),no).
sam(sam2f,[b,c],q_abce_4,fr(1:3/8,1/2),fr(1:14/16),no).
sam(sam2f,[b,c],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam2f,[c],q_acef_1,fr(1:1/3,2/3),fr(1:3/3),no).
sam(sam2f,[c],q_acef_2,fr(1:1/5,4/5),fr(1:5/5),no).
sam(sam2f,[c],q_cdef_1,fr(1/3,5/3),fr(6/3),no).
sam(sam2f,[c],q_cdef_2,fr(3/2,5/2),fr(8/2),no).
sam(sam2f,[b,c],q_bcde_1,fr(3/8,3/4),fr(36/32),no).
sam(sam2f,[b,c],q_bcde_2,fr(2/3,5/6),fr(27/18),no).
sam(sam2f,[b,c],q_bcde_3,fr(1/8,7/4),fr(60/32),no).
sam(sam2f,[b,c],q_bcde_4,fr(1/2,7/6),fr(20/12),no).
sam(sam2f,[b,c],q_bcde_5,fr(5/3,7/6),fr(51/18),no).
sam(sam2f,[b,c],q_bcde_6,fr(4/3,7/6),fr(45/18),no).
sam(sam2f,[b,c],q_bcde_7,fr(3/8,5/6),fr(58/48),no).
sam(sam2f,[b,c],q_bcde_8,fr(4/5,3/10),fr(55/50),no).
sam(sam2f,[b,c],q_bcde_9,fr(7/8,3/4),fr(52/32),no).
sam(sam2f,[b,c],q_bcdf_1,fr(3/6,2/4),fr(24/24),no).
sam(sam2f,[b,c],q_bcdf_2,fr(5/10,4/8),fr(80/80),no).
sam(sam2f,[c],q_acde_1,fr(1:1/8,1:3/8),fr(2:4/8),no).
sam(sam2f,[c],q_acde_2,fr(1:1/6,2:1/6),fr(3:2/6),no).
sam(sam2f,[b,c],q_abcde_1,fr(1/3,1:5/6),fr(1:21/18),no).
sam(sam2f,[b,c],q_abcde_2,fr(1/4,2:1/12),fr(2:16/48),no).
```

Logfile: log3a

simStudents([[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]).

```
sam(sam3a,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam3a,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam3a,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam3a,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam3a,[b,c],q_bc_3,fr(5/8,1/6),fr(38/48),no).
sam(sam3a,[c],q_cd_1,fr(4/9,2/9),fr(6/9),no).
sam(sam3a,[c,mg],q_cd_2,fr(12/64,4/64),fr(1/4),ok).
sam(sam3a,[c,mg],q_cd_3,fr(9/24,3/24),fr(1/4),no).
sam(sam3a,[c,mg],q_cd_4,fr(9/16,3/16),fr(2/6),no).
sam(sam3a,[c],q_cd_5,fr(1/6,1/6),fr(2/6),no).
sam(sam3a,[c],q_cd_6,fr(3/8,1/8),fr(4/8),no).
sam(sam3a,[c,e],q_ce_1,fr(5/7,6/7),fr(1:4/7),ok).
sam(sam3a,[c,e],q_ce_2,fr(4/7,8/7),fr(1:5/7),ok).
sam(sam3a,[c,e],q_ce_3,fr(8/5,6/5),fr(2:4/5),ok).
sam(sam3a,[c,e],q_ce_4,fr(2/3,2/3),fr(1:1/3),ok).
sam(sam3a,[c,e],q_ce_5,fr(5/6,2/6),fr(1:1/6),ok).
sam(sam3a,[c,e],q_ce_6,fr(7/5,2/5),fr(1:4/5),ok).
sam(sam3a,[a,c,e],q_ace_1,fr(1:1/5,2/5),fr(1:3/5),ok).
sam(sam3a,[a,c,e],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam3a,[a,c,e],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam3a,[b,c],q_bcd_1,fr(1/2,1/4),fr(6/8),no).
sam(sam3a,[b,c],q_bcd_2,fr(5/6,1/8),fr(46/48),no).
sam(sam3a,[b,c],q_bcd_3,fr(1/2,1/6),fr(8/12),no).
sam(sam3a,[b,c],q_bcd_4,fr(1/12,1/6),fr(18/72),no).
sam(sam3a,[b,c],q_bcd_5,fr(2/15,1/5),fr(25/75),no).
sam(sam3a,[b,c,e],q_bce_1,fr(2/3,3/5),fr(1:4/15),ok).
sam(sam3a,[b,c,e],q_bce_2,fr(1/2,7/3),fr(2:5/6),ok).
sam(sam3a,[b,c,e],q_bce_3,fr(5/2,8/5),fr(4:1/10),ok).
sam(sam3a,[b,c,e],q_bce_4,fr(4/5,3/4),fr(1:11/20),ok).
sam(sam3a,[b,c,e],q_bce_5,fr(5/6,1/3),fr(1:3/18),no).
sam(sam3a,[b,c,e],q_bce_6,fr(6/7,3/8),fr(1:13/56),ok).
sam(sam3a,[b,c,e],q_bce_7,fr(6/7,5/8),fr(1:27/56),ok).
sam(sam3a,[b,c,e],q_bce_8,fr(5/8,4/5),fr(1:17/40),ok).
sam(sam3a,[b,c,e],q_bce_9,fr(5/8,5/6),fr(1:22/48),no).
sam(sam3a,[b,c,e],q_bce_10,fr(3/8,5/6),fr(1:10/48),no).
sam(sam3a,[b,c,e],q_bce_11,fr(8/9,3/5),fr(1:22/45),ok).
sam(sam3a,[c,e],q_cde_1,fr(5/8,9/8),fr(1:6/8),no).
sam(sam3a,[c,e],q_cde_2,fr(8/9,4/9),fr(1:3/9),no).
sam(sam3a,[c,e],q_cde_3,fr(5/4,5/4),fr(2:2/4),no).
sam(sam3a,[c],q_cdf_1,fr(5/7,2/7),fr(7/7),no).
sam(sam3a,[c],q_cdf_2,fr(4/5,1/5),fr(5/5),no).
sam(sam3a,[c],q_cdf_3,fr(1/2,1/2),fr(2/2),no).
sam(sam3a,[c],q_cdf_4,fr(4/5,1/5),fr(5/5),no).
sam(sam3a,[c],q_cdf_5,fr(1/2,1/2),fr(2/2),no).
sam(sam3a,[a,b,c,e],q_abce_1,fr(1:2/3,1:3/5),fr(3:4/15),ok).
sam(sam3a,[a,b,c,e],q_abce_2,fr(2:1/2,7/3),fr(4:5/6),ok).
sam(sam3a,[a,b,c,e],q_abce_3,fr(1/4,2:1/8),fr(2:12/32),no).
sam(sam3a,[a,b,c,e],q_abce_4,fr(1:3/8,1/2),fr(1:14/16),no).
sam(sam3a,[a,b,c,e],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam3a,[a,c,e],q_acef_1,fr(1:1/3,2/3),fr(2:0/3),no).
sam(sam3a,[a,c,e],q_acef_2,fr(1:1/5,4/5),fr(2:0/5),no).
sam(sam3a,[c,e],q_cdef_1,fr(1/3,5/3),fr(2:0/3),no).
sam(sam3a,[c,e],q_cdef_2,fr(3/2,5/2),fr(4:0/2),no).
sam(sam3a,[b,c,e],q_bcde_1,fr(3/8,3/4),fr(1:4/32),no).
sam(sam3a,[b,c,e],q_bcde_2,fr(2/3,5/6),fr(1:9/18),no).
sam(sam3a,[b,c,e],q_bcde_3,fr(1/8,7/4),fr(1:28/32),no).
sam(sam3a,[b,c,e],q_bcde_4,fr(1/2,7/6),fr(1:8/12),no).
sam(sam3a,[b,c,e],q_bcde_5,fr(5/3,7/6),fr(2:15/18),no).
sam(sam3a,[b,c,e],q_bcde_6,fr(4/3,7/6),fr(2:9/18),no).
sam(sam3a,[b,c,e],q_bcde_7,fr(3/8,5/6),fr(1:10/48),no).
sam(sam3a,[b,c,e],q_bcde_8,fr(4/5,3/10),fr(1:5/50),no).
sam(sam3a,[b,c,e],q_bcde_9,fr(7/8,3/4),fr(1:20/32),no).
sam(sam3a,[b,c],q_bcdf_1,fr(3/6,2/4),fr(24/24),no).
sam(sam3a,[b,c],q_bcdf_2,fr(5/10,4/8),fr(80/80),no).
sam(sam3a,[a,c,e],q_acde_1,fr(1:1/8,1:3/8),fr(2:4/8),no).
sam(sam3a,[a,c,e],q_acde_2,fr(1:1/6,2:1/6),fr(3:2/6),no).
sam(sam3a,[a,b,c,e],q_abcde_1,fr(1/3,1:5/6),fr(2:3/18),no).
sam(sam3a,[a,b,c,e],q_abcde_2,fr(1/4,2:1/12),fr(2:16/48),no).
```

Logfile: log4a

simStudents([[sam4a-[[makeCommon],[checkAndAdd],[]]]]).

```
sam(sam4a,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam4a,[c],q_c_2,fr(3/5,1/5),fr(4/5),ok).
sam(sam4a,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam4a,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam4a,[b,c],q_bc_3,fr(5/8,1/6),fr(38/48),no).
sam(sam4a,[c],q_cd_1,fr(4/9,2/9),fr(2/3),ok).              %tweaked
sam(sam4a,[c],q_cd_2,fr(12/64,4/64),fr(1/4),ok).           %tweaked
sam(sam4a,[c],q_cd_3,fr(9/24,3/24),fr(12/24),no).
sam(sam4a,[c],q_cd_4,fr(9/16,3/16),fr(12/16),no).
sam(sam4a,[c],q_cd_5,fr(1/6,1/6),fr(2/6),no).
sam(sam4a,[c],q_cd_6,fr(3/8,1/8),fr(4/8),no).
sam(sam4a,[c],q_ce_1,fr(5/7,6/7),fr(1:4/7),ok).            %tweaked
sam(sam4a,[c],q_ce_2,fr(4/7,8/7),fr(1:5/7),ok).            %tweaked
sam(sam4a,[c],q_ce_3,fr(8/5,6/5),fr(14/5),no).
sam(sam4a,[c],q_ce_4,fr(2/3,2/3),fr(4/3),no).
sam(sam4a,[c],q_ce_5,fr(5/6,2/6),fr(7/6),no).
sam(sam4a,[c],q_ce_6,fr(7/5,2/5),fr(9/5),no).
sam(sam4a,[c],q_ace_1,fr(1:1/5,2/5),fr(1:3/5),ok).
sam(sam4a,[c],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam4a,[c],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam4a,[b,c],q_bcd_1,fr(1/2,1/4),fr(3/4),ok).              %tweaked
sam(sam4a,[b,c],q_bcd_2,fr(5/6,1/8),fr(23/24),ok).            %tweaked
sam(sam4a,[b,c],q_bcd_3,fr(1/2,1/6),fr(8/12),no).
sam(sam4a,[b,c],q_bcd_4,fr(1/12,1/6),fr(18/72),no).
sam(sam4a,[b,c],q_bcd_5,fr(2/15,1/5),fr(25/75),no).
sam(sam4a,[b,c],q_bce_1,fr(2/3,3/5),fr(1:4/15),ok).           %tweaked
sam(sam4a,[b,c],q_bce_2,fr(1/2,7/3),fr(2:5/6),ok).            %tweaked
sam(sam4a,[b,c],q_bce_3,fr(5/2,8/5),fr(41/10),no).
sam(sam4a,[b,c],q_bce_4,fr(4/5,3/4),fr(31/20),no).
sam(sam4a,[b,c],q_bce_5,fr(5/6,1/3),fr(21/18),no).
sam(sam4a,[b,c],q_bce_6,fr(6/7,3/8),fr(69/56),no).
sam(sam4a,[b,c],q_bce_7,fr(6/7,5/8),fr(83/56),no).
sam(sam4a,[b,c],q_bce_8,fr(5/8,4/5),fr(57/40),no).
sam(sam4a,[b,c],q_bce_9,fr(5/8,5/6),fr(70/48),no).
sam(sam4a,[b,c],q_bce_10,fr(3/8,5/6),fr(58/48),no).
sam(sam4a,[b,c],q_bce_11,fr(8/9,3/5),fr(67/45),no).
sam(sam4a,[c],q_cde_1,fr(5/8,9/8),fr(1:3/4),ok).              %tweaked
sam(sam4a,[c],q_cde_2,fr(8/9,4/9),fr(1:1/3),ok).              %tweaked
sam(sam4a,[c],q_cde_3,fr(5/4,5/4),fr(10/4),no).
sam(sam4a,[c],q_cdf_1,fr(5/7,2/7),fr(1),ok).                  %tweaked
sam(sam4a,[c],q_cdf_2,fr(4/5,1/5),fr(1),ok).                  %tweaked
sam(sam4a,[c],q_cdf_3,fr(1/2,1/2),fr(2/2),no).
sam(sam4a,[c],q_cdf_4,fr(4/5,1/5),fr(5/5),no).
sam(sam4a,[c],q_cdf_5,fr(1/2,1/2),fr(2/2),no).
sam(sam4a,[b,c],q_abce_1,fr(1:2/3,1:3/5),fr(3:4/15),ok).      %tweaked
sam(sam4a,[b,c],q_abce_2,fr(2:1/2,7/3),fr(4:5/6),ok).         %tweaked
sam(sam4a,[b,c],q_abce_3,fr(1/4,2:1/8),fr(2:12/32),no).
sam(sam4a,[b,c],q_abce_4,fr(1:3/8,1/2),fr(1:14/16),no).
sam(sam4a,[b,c],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam4a,[c],q_acef_1,fr(1:1/3,2/3),fr(2),ok).               %tweaked
sam(sam4a,[c],q_acef_2,fr(1:1/5,4/5),fr(2),ok).               %tweaked
sam(sam4a,[c],q_cdef_1,fr(1/3,5/3),fr(2),ok).                 %tweaked
sam(sam4a,[c],q_cdef_2,fr(3/2,5/2),fr(4),ok).                 %tweaked
sam(sam4a,[b,c],q_bcde_1,fr(3/8,3/4),fr(1:1/8),ok).           %tweaked
sam(sam4a,[b,c],q_bcde_2,fr(2/3,5/6),fr(1:1/2),ok).           %tweaked
sam(sam4a,[b,c],q_bcde_3,fr(1/8,7/4),fr(60/32),no).
sam(sam4a,[b,c],q_bcde_4,fr(1/2,7/6),fr(20/12),no).
sam(sam4a,[b,c],q_bcde_5,fr(5/3,7/6),fr(51/18),no).
sam(sam4a,[b,c],q_bcde_6,fr(4/3,7/6),fr(45/18),no).
sam(sam4a,[b,c],q_bcde_7,fr(3/8,5/6),fr(58/48),no).
sam(sam4a,[b,c],q_bcde_8,fr(4/5,3/10),fr(55/50),no).
sam(sam4a,[b,c],q_bcde_9,fr(7/8,3/4),fr(52/32),no).
sam(sam4a,[b,c],q_bcdf_1,fr(3/6,2/4),fr(1),ok).               %tweaked
sam(sam4a,[b,c],q_bcdf_2,fr(5/10,4/8),fr(1),ok).              %tweaked
sam(sam4a,[c],q_acde_1,fr(1:1/8,1:3/8),fr(2:1/2),ok).         %tweaked
sam(sam4a,[c],q_acde_2,fr(1:1/6,2:1/6),fr(3:1/3),ok).         %tweaked
sam(sam4a,[b,c],q_abcde_1,fr(1/3,1:5/6),fr(2:1/6),ok).        %tweaked
sam(sam4a,[b,c],q_abcde_2,fr(1/4,2:1/12),fr(2:1/3),ok).       %tweaked
```

Logfile: log5a

simStudents([[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]).

```
sam(sam1a,[c],q_c_1,fr(1/3,1/3),fr(2/3),ok).
sam(sam1a,[c],q_c_2,fr(3/5,1/5),fr(x),no).                    %tweaked
sam(sam1a,[b,c],q_bc_1,fr(1/2,1/5),fr(7/10),ok).
sam(sam1a,[b,c],q_bc_2,fr(1/3,1/5),fr(8/15),ok).
sam(sam1a,[b,c,d],q_bc_3,fr(5/8,1/6),fr(19/24),ok).
sam(sam1a,[c,d],q_cd_1,fr(4/9,2/9),fr(x),no).                 %tweaked
sam(sam1a,[c,d],q_cd_2,fr(12/64,4/64),fr(1/4),ok).
sam(sam1a,[c,d],q_cd_3,fr(9/24,3/24),fr(1/2),ok).
sam(sam1a,[c,d],q_cd_4,fr(9/16,3/16),fr(3/4),ok).
sam(sam1a,[c,d],q_cd_5,fr(1/6,1/6),fr(1/3),ok).
sam(sam1a,[c,d],q_cd_6,fr(3/8,1/8),fr(1/2),ok).
sam(sam1a,[c,e],q_ce_1,fr(5/7,6/7),fr(x),no).                 %tweaked
sam(sam1a,[c,e],q_ce_2,fr(4/7,8/7),fr(1:5/7),ok).
sam(sam1a,[c,e],q_ce_3,fr(8/5,6/5),fr(2:4/5),ok).
sam(sam1a,[c,e],q_ce_4,fr(2/3,2/3),fr(1:1/3),ok).
sam(sam1a,[c,e],q_ce_5,fr(5/6,2/6),fr(1:1/6),ok).
sam(sam1a,[c,e],q_ce_6,fr(7/5,2/5),fr(1:4/5),ok).
sam(sam1a,[a,c,e],q_ace_1,fr(1:1/5,2/5),fr(x),no).            %tweaked
sam(sam1a,[a,c,e],q_ace_2,fr(2:3/7,2/7),fr(2:5/7),ok).
sam(sam1a,[a,c,e],q_ace_3,fr(1:1/5,3/5),fr(1:4/5),ok).
sam(sam1a,[b,c,d],q_bcd_1,fr(1/2,1/4),fr(3/4),ok).
sam(sam1a,[b,c,d],q_bcd_2,fr(5/6,1/8),fr(23/24),ok).
sam(sam1a,[b,c,d],q_bcd_3,fr(1/2,1/6),fr(2/3),ok).
sam(sam1a,[b,c,d],q_bcd_4,fr(1/12,1/6),fr(1/4),ok).
sam(sam1a,[b,c,d],q_bcd_5,fr(2/15,1/5),fr(1/3),ok).
sam(sam1a,[b,c,e],q_bce_1,fr(2/3,3/5),fr(x),no).              %tweaked
sam(sam1a,[b,c,e],q_bce_2,fr(1/2,7/3),fr(2:5/6),ok).
sam(sam1a,[b,c,e],q_bce_3,fr(5/2,8/5),fr(4:1/10),ok).
sam(sam1a,[b,c,e],q_bce_4,fr(4/5,3/4),fr(1:11/20),ok).
sam(sam1a,[b,c,d,e],q_bce_5,fr(5/6,1/3),fr(1:1/6),ok).
sam(sam1a,[b,c,e],q_bce_6,fr(6/7,3/8),fr(1:13/56),ok).
sam(sam1a,[b,c,e],q_bce_7,fr(6/7,5/8),fr(1:27/56),ok).
sam(sam1a,[b,c,e],q_bce_8,fr(5/8,4/5),fr(1:17/40),ok).
sam(sam1a,[b,c,d,e],q_bce_9,fr(5/8,5/6),fr(1:11/24),ok).
sam(sam1a,[b,c,d,e],q_bce_10,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam1a,[b,c,e],q_bce_11,fr(8/9,3/5),fr(1:22/45),ok).
sam(sam1a,[c,d,e],q_cde_1,fr(5/8,9/8),fr(1:3/4),ok).
sam(sam1a,[c,d,e],q_cde_2,fr(8/9,4/9),fr(1:1/3),ok).
sam(sam1a,[c,d,e],q_cde_3,fr(5/4,5/4),fr(2:1/2),ok).
sam(sam1a,[c,d,f],q_cdf_1,fr(5/7,2/7),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_2,fr(4/5,1/5),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_3,fr(1/2,1/2),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_4,fr(4/5,1/5),fr(1),ok).
sam(sam1a,[c,d,f],q_cdf_5,fr(1/2,1/2),fr(1),ok).
sam(sam1a,[a,b,c,e],q_abce_1,fr(1:2/3,1:3/5),fr(3:4/15),ok).
sam(sam1a,[a,b,c,e],q_abce_2,fr(2:1/2,7/3),fr(4:5/6),ok).
sam(sam1a,[a,b,c,d,e],q_abce_3,fr(1/4,2:1/8),fr(2:3/8),ok).
sam(sam1a,[a,b,c,d,e],q_abce_4,fr(1:3/8,1/2),fr(1:7/8),ok).
sam(sam1a,[a,b,c,e],q_abce_5,fr(1/5,1:5/7),fr(1:32/35),ok).
sam(sam1a,[a,c,d,e,f],q_acef_1,fr(1:1/3,2/3),fr(2),ok).
sam(sam1a,[a,c,d,e,f],q_acef_2,fr(1:1/5,4/5),fr(2),ok).
sam(sam1a,[c,d,e,f],q_cdef_1,fr(1/3,5/3),fr(2),ok).
sam(sam1a,[c,d,e,f],q_cdef_2,fr(3/2,5/2),fr(4),ok).
sam(sam1a,[b,c,d,e],q_bcde_1,fr(3/8,3/4),fr(1:1/8),ok).
sam(sam1a,[b,c,d,e],q_bcde_2,fr(2/3,5/6),fr(1:1/2),ok).
sam(sam1a,[b,c,d,e],q_bcde_3,fr(1/8,7/4),fr(1:7/8),ok).
sam(sam1a,[b,c,d,e],q_bcde_4,fr(1/2,7/6),fr(1:2/3),ok).
sam(sam1a,[b,c,d,e],q_bcde_5,fr(5/3,7/6),fr(2:5/6),ok).
sam(sam1a,[b,c,d,e],q_bcde_6,fr(4/3,7/6),fr(2:1/2),ok).
sam(sam1a,[b,c,d,e],q_bcde_7,fr(3/8,5/6),fr(1:5/24),ok).
sam(sam1a,[b,c,d,e],q_bcde_8,fr(4/5,3/10),fr(1:1/10),ok).
sam(sam1a,[b,c,d,e],q_bcde_9,fr(7/8,3/4),fr(1:5/8),ok).
sam(sam1a,[b,c,d,f],q_bcdf_1,fr(3/6,2/4),fr(1),ok).
sam(sam1a,[b,c,d,f],q_bcdf_2,fr(5/10,4/8),fr(1),ok).
sam(sam1a,[a,c,d,e],q_acde_1,fr(1:1/8,1:3/8),fr(2:1/2),ok).
sam(sam1a,[a,c,d,e],q_acde_2,fr(1:1/6,2:1/6),fr(3:1/3),ok).
sam(sam1a,[a,b,c,d,e],q_abcde_1,fr(1/3,1:5/6),fr(2:1/6),ok).
sam(sam1a,[a,b,c,d,e],q_abcde_2,fr(1/4,2:1/12),fr(2:1/3),ok).
```

Appendix J. Running XP Adaptive Test

This appendix contains the results from running XP for a selected list of simulated students. This was described in Section 6.5. The following legend to represent the different skills:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole

% XP ADAPTIVE TEST output xp_1a

```
Student=[[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 4
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).

Selected Node : 5
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).

Selected Node : 6
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).

% Summary - XP ADAPTIVE TEST output
problems_presented(7,68).
opportunities_presented([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
opportunities_correctly_applied([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
```

% XP ADAPTIVE TEST output xp_2e

Student = [[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
```

Selected Node : 3

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
```

% Summary - XP1 ADAPTIVE TEST output

```
problems_presented(11,68).
opportunities_presented([(a,4),(b,5),(c,11),(d,7),(e,8),(f,4)]).
opportunities_correctly_applied([(a,3),(b,4),(c,7),(d,4),(e,6),(f,0)]).
```

% XP ADAPTIVE TEST output xp_2f

Student = [[sam2f-[[makeCommon],[checkAndAdd],[]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
```

Selected Node : 3

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
```

Selected Node : 2

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
```

Selected Node : 1

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
```

% Summary - XP1 ADAPTIVE TEST output

```
problems_presented(15,68).
opportunities_presented([(a,4),(b,6),(c,15),(d,8),(e,9),(f,4)]).
opportunities_correctly_applied([(a,1),(b,1),(c,3),(d,0),(e,1),(f,0)]).
```

% XP ADAPTIVE TEST output xp_3a

Student = [[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
```

Selected Node : 3

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
```

Selected Node : 2

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), correct).
```

% Summary - XP1 ADAPTIVE TEST output

```
problems_presented(14,68).
opportunities_presented([(a,4),(b,6),(c,14),(d,8),(e,9),(f,4)]).
opportunities_correctly_applied([(a,2),(b,3),(c,5),(d,0),(e,4),(f,0)]).
```

% XP ADAPTIVE TEST output xp_4a

Student = [[sam4a-[[makeCommon],[checkAndAdd],[]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
```

Selected Node : 5

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
```

Selected Node : 6

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
```

% Summary - XP1 ADAPTIVE TEST output

```
problems_presented(7,68).
opportunities_presented([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
opportunities_correctly_applied([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
```

% XP ADAPTIVE TEST output xp_5a

Student = [[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

```
Selected Node : 4
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).

Selected Node : 5
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).

Selected Node : 6
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
```

% Summary - XP1 ADAPTIVE TEST output

```
problems_presented(7,68).
opportunities_presented([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
opportunities_correctly_applied([(a,4),(b,4),(c,7),(d,5),(e,6),(f,3)]).
```

Appendix K. Running ST Sequential Test

This appendix contains the results from running ST for a selected list of simulated students. This was described in Section 6.5. The following legend to represent the different skills:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole

% ST ADAPTIVE TEST output st_1a

Student = [[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], ok).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], ok).
visited(q_cd_1, [c,d], ok).
visited(q_cd_2, [c,d], ok).
visited(q_cd_3, [c,d], ok).
visited(q_cd_4, [c,d], ok).
visited(q_cd_5, [c,d], ok).
visited(q_cd_6, [c,d], ok).
visited(q_ce_1, [c,e], ok).
visited(q_ce_2, [c,e], ok).
visited(q_ce_3, [c,e], ok).
visited(q_ce_4, [c,e], ok).
visited(q_ce_5, [c,e], ok).
visited(q_ce_6, [c,e], ok).
visited(q_ace_1, [a,c,e], ok).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], ok).
visited(q_bcd_2, [b,c,d], ok).
visited(q_bcd_3, [b,c,d], ok).
visited(q_bcd_4, [b,c,d], ok).
visited(q_bcd_5, [b,c,d], ok).
visited(q_bce_1, [b,c,e], ok).
visited(q_bce_2, [b,c,e], ok).
visited(q_bce_3, [b,c,e], ok).
visited(q_bce_4, [b,c,e], ok).
visited(q_bce_5, [b,c,d,e], ok).
visited(q_bce_6, [b,c,e], ok).
visited(q_bce_7, [b,c,e], ok).
visited(q_bce_8, [b,c,e], ok).
visited(q_bce_9, [b,c,d,e], ok).
visited(q_bce_10, [b,c,d,e], ok).
visited(q_bce_11, [b,c,e], ok).
visited(q_cde_1, [c,d,e], ok).
visited(q_cde_2, [c,d,e], ok).
visited(q_cde_3, [c,d,e], ok).
visited(q_cdf_1, [c,d,f], ok).
visited(q_cdf_2, [c,d,f], ok).
visited(q_cdf_3, [c,d,f], ok).
visited(q_cdf_4, [c,d,f], ok).
visited(q_cdf_5, [c,d,f], ok).
visited(q_abce_1, [a,b,c,e], ok).
visited(q_abce_2, [a,b,c,e], ok).
visited(q_abce_3, [a,b,c,d,e], ok).
visited(q_abce_4, [a,b,c,d,e], ok).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], ok).
visited(q_acef_2, [a,c,d,e,f], ok).
visited(q_cdef_1, [c,d,e,f], ok).
visited(q_cdef_2, [c,d,e,f], ok).
visited(q_bcde_1, [b,c,d,e], ok).
visited(q_bcde_2, [b,c,d,e], ok).
visited(q_bcde_3, [b,c,d,e], ok).
visited(q_bcde_4, [b,c,d,e], ok).
visited(q_bcde_5, [b,c,d,e], ok).
visited(q_bcde_6, [b,c,d,e], ok).
visited(q_bcde_7, [b,c,d,e], ok).
visited(q_bcde_8, [b,c,d,e], ok).
visited(q_bcde_9, [b,c,d,e], ok).
visited(q_bcdf_1, [b,c,d,f], ok).
visited(q_bcdf_2, [b,c,d,f], ok).
visited(q_acde_1, [a,c,d,e], ok).
visited(q_acde_2, [a,c,d,e], ok).
visited(q_abcde_1, [a,b,c,d,e], ok).
visited(q_abcde_2, [a,b,c,d,e], ok).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
```

% ST ADAPTIVE TEST output st_2e

Student = [[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], ok).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], ok).
visited(q_cd_1, [c,d], ok).
visited(q_cd_2, [c,d], ok).
visited(q_cd_3, [c,d], ok).
visited(q_cd_4, [c,d], ok).
visited(q_cd_5, [c,d], ok).
visited(q_cd_6, [c,d], ok).
visited(q_ce_1, [c,e], ok).
visited(q_ce_2, [c,e], ok).
visited(q_ce_3, [c,e], ok).
visited(q_ce_4, [c,e], ok).
visited(q_ce_5, [c,e], ok).
visited(q_ce_6, [c,e], ok).
visited(q_ace_1, [a,c,e], ok).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], ok).
visited(q_bcd_2, [b,c,d], ok).
visited(q_bcd_3, [b,c,d], ok).
visited(q_bcd_4, [b,c,d], ok).
visited(q_bcd_5, [b,c,d], ok).
visited(q_bce_1, [b,c,e], ok).
visited(q_bce_2, [b,c,e], ok).
visited(q_bce_3, [b,c,e], ok).
visited(q_bce_4, [b,c,e], ok).
visited(q_bce_5, [b,c,d,e], ok).
visited(q_bce_6, [b,c,e], ok).
visited(q_bce_7, [b,c,e], ok).
visited(q_bce_8, [b,c,e], ok).
visited(q_bce_9, [b,c,d,e], ok).
visited(q_bce_10, [b,c,d,e], ok).
visited(q_bce_11, [b,c,e], ok).
visited(q_cde_1, [c,d,e], ok).
visited(q_cde_2, [c,d,e], ok).
visited(q_cde_3, [c,d,e], ok).
visited(q_cdf_1, [c,d,f], no).
visited(q_cdf_2, [c,d,f], no).
visited(q_cdf_3, [c,d,f], no).
visited(q_cdf_4, [c,d,f], no).
visited(q_cdf_5, [c,d,f], no).
visited(q_abce_1, [a,b,c,e], ok).
visited(q_abce_2, [a,b,c,e], ok).
visited(q_abce_3, [a,b,c,d,e], ok).
visited(q_abce_4, [a,b,c,d,e], ok).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], no).
visited(q_acef_2, [a,c,d,e,f], no).
visited(q_cdef_1, [c,d,e,f], no).
visited(q_cdef_2, [c,d,e,f], no).
visited(q_bcde_1, [b,c,d,e], ok).
visited(q_bcde_2, [b,c,d,e], ok).
visited(q_bcde_3, [b,c,d,e], ok).
visited(q_bcde_4, [b,c,d,e], ok).
visited(q_bcde_5, [b,c,d,e], ok).
visited(q_bcde_6, [b,c,d,e], ok).
visited(q_bcde_7, [b,c,d,e], ok).
visited(q_bcde_8, [b,c,d,e], ok).
visited(q_bcde_9, [b,c,d,e], ok).
visited(q_bcdf_1, [b,c,d,f], no).
visited(q_bcdf_2, [b,c,d,f], no).
visited(q_acde_1, [a,c,d,e], ok).
visited(q_acde_2, [a,c,d,e], ok).
visited(q_abcde_1, [a,b,c,d,e], ok).
visited(q_abcde_2, [a,b,c,d,e], ok).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,12),(b,35),(c,57),(d,33),(e,41),(f,0)]).
```

% ST ADAPTIVE TEST output st_2f

Student = [[sam2f-[[makeCommon],[checkAndAdd],[]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], ok).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], no).
visited(q_cd_1, [c,d], no).
visited(q_cd_2, [c,d], no).
visited(q_cd_3, [c,d], no).
visited(q_cd_4, [c,d], no).
visited(q_cd_5, [c,d], no).
visited(q_cd_6, [c,d], no).
visited(q_ce_1, [c,e], no).
visited(q_ce_2, [c,e], no).
visited(q_ce_3, [c,e], no).
visited(q_ce_4, [c,e], no).
visited(q_ce_5, [c,e], no).
visited(q_ce_6, [c,e], no).
visited(q_ace_1, [a,c,e], ok).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], no).
visited(q_bcd_2, [b,c,d], no).
visited(q_bcd_3, [b,c,d], no).
visited(q_bcd_4, [b,c,d], no).
visited(q_bcd_5, [b,c,d], no).
visited(q_bce_1, [b,c,e], no).
visited(q_bce_2, [b,c,e], no).
visited(q_bce_3, [b,c,e], no).
visited(q_bce_4, [b,c,e], no).
visited(q_bce_5, [b,c,d,e], no).
visited(q_bce_6, [b,c,e], no).
visited(q_bce_7, [b,c,e], no).
visited(q_bce_8, [b,c,e], no).
visited(q_bce_9, [b,c,d,e], no).
visited(q_bce_10, [b,c,d,e], no).
visited(q_bce_11, [b,c,e], no).
visited(q_cde_1, [c,d,e], no).
visited(q_cde_2, [c,d,e], no).
visited(q_cde_3, [c,d,e], no).
visited(q_cdf_1, [c,d,f], no).
visited(q_cdf_2, [c,d,f], no).
visited(q_cdf_3, [c,d,f], no).
visited(q_cdf_4, [c,d,f], no).
visited(q_cdf_5, [c,d,f], no).
visited(q_abce_1, [a,b,c,e], no).
visited(q_abce_2, [a,b,c,e], no).
visited(q_abce_3, [a,b,c,d,e], no).
visited(q_abce_4, [a,b,c,d,e], no).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], no).
visited(q_acef_2, [a,c,d,e,f], no).
visited(q_cdef_1, [c,d,e,f], no).
visited(q_cdef_2, [c,d,e,f], no).
visited(q_bcde_1, [b,c,d,e], no).
visited(q_bcde_2, [b,c,d,e], no).
visited(q_bcde_3, [b,c,d,e], no).
visited(q_bcde_4, [b,c,d,e], no).
visited(q_bcde_5, [b,c,d,e], no).
visited(q_bcde_6, [b,c,d,e], no).
visited(q_bcde_7, [b,c,d,e], no).
visited(q_bcde_8, [b,c,d,e], no).
visited(q_bcde_9, [b,c,d,e], no).
visited(q_bcdf_1, [b,c,d,f], no).
visited(q_bcdf_2, [b,c,d,f], no).
visited(q_acde_1, [a,c,d,e], no).
visited(q_acde_2, [a,c,d,e], no).
visited(q_abcde_1, [a,b,c,d,e], no).
visited(q_abcde_2, [a,b,c,d,e], no).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,4),(b,3),(c,8),(d,0),(e,4),(f,0)]).
```

% ST ADAPTIVE TEST output st_3a

Student = [[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], ok).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], no).
visited(q_cd_1, [c,d], no).
visited(q_cd_2, [c,d], ok).
visited(q_cd_3, [c,d], no).
visited(q_cd_4, [c,d], no).
visited(q_cd_5, [c,d], no).
visited(q_cd_6, [c,d], no).
visited(q_ce_1, [c,e], ok).
visited(q_ce_2, [c,e], ok).
visited(q_ce_3, [c,e], ok).
visited(q_ce_4, [c,e], ok).
visited(q_ce_5, [c,e], ok).
visited(q_ce_6, [c,e], ok).
visited(q_ace_1, [a,c,e], ok).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], no).
visited(q_bcd_2, [b,c,d], no).
visited(q_bcd_3, [b,c,d], no).
visited(q_bcd_4, [b,c,d], no).
visited(q_bcd_5, [b,c,d], no).
visited(q_bce_1, [b,c,e], ok).
visited(q_bce_2, [b,c,e], ok).
visited(q_bce_3, [b,c,e], ok).
visited(q_bce_4, [b,c,e], ok).
visited(q_bce_5, [b,c,d,e], no).
visited(q_bce_6, [b,c,e], ok).
visited(q_bce_7, [b,c,e], ok).
visited(q_bce_8, [b,c,e], ok).
visited(q_bce_9, [b,c,d,e], no).
visited(q_bce_10, [b,c,d,e], no).
visited(q_bce_11, [b,c,e], ok).
visited(q_cde_1, [c,d,e], no).
visited(q_cde_2, [c,d,e], no).
visited(q_cde_3, [c,d,e], no).
visited(q_cdf_1, [c,d,f], no).
visited(q_cdf_2, [c,d,f], no).
visited(q_cdf_3, [c,d,f], no).
visited(q_cdf_4, [c,d,f], no).
visited(q_cdf_5, [c,d,f], no).
visited(q_abce_1, [a,b,c,e], ok).
visited(q_abce_2, [a,b,c,e], ok).
visited(q_abce_3, [a,b,c,d,e], no).
visited(q_abce_4, [a,b,c,d,e], no).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], no).
visited(q_acef_2, [a,c,d,e,f], no).
visited(q_cdef_1, [c,d,e,f], no).
visited(q_cdef_2, [c,d,e,f], no).
visited(q_bcde_1, [b,c,d,e], no).
visited(q_bcde_2, [b,c,d,e], no).
visited(q_bcde_3, [b,c,d,e], no).
visited(q_bcde_4, [b,c,d,e], no).
visited(q_bcde_5, [b,c,d,e], no).
visited(q_bcde_6, [b,c,d,e], no).
visited(q_bcde_7, [b,c,d,e], no).
visited(q_bcde_8, [b,c,d,e], no).
visited(q_bcde_9, [b,c,d,e], no).
visited(q_bcdf_1, [b,c,d,f], no).
visited(q_bcdf_2, [b,c,d,f], no).
visited(q_acde_1, [a,c,d,e], no).
visited(q_acde_2, [a,c,d,e], no).
visited(q_abcde_1, [a,b,c,d,e], no).
visited(q_abcde_2, [a,b,c,d,e], no).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,6),(b,13),(c,25),(d,1),(e,20),(f,0)]).
```

% ST ADAPTIVE TEST output st_4a

Student = [[sam4a-[[makeCommon],[checkAndAdd],[]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], ok).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], no).
visited(q_cd_1, [c,d], ok).
visited(q_cd_2, [c,d], ok).
visited(q_cd_3, [c,d], no).
visited(q_cd_4, [c,d], no).
visited(q_cd_5, [c,d], no).
visited(q_cd_6, [c,d], no).
visited(q_ce_1, [c,e], ok).
visited(q_ce_2, [c,e], ok).
visited(q_ce_3, [c,e], no).
visited(q_ce_4, [c,e], no).
visited(q_ce_5, [c,e], no).
visited(q_ce_6, [c,e], no).
visited(q_ace_1, [a,c,e], ok).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], ok).
visited(q_bcd_2, [b,c,d], ok).
visited(q_bcd_3, [b,c,d], no).
visited(q_bcd_4, [b,c,d], no).
visited(q_bcd_5, [b,c,d], no).
visited(q_bce_1, [b,c,e], ok).
visited(q_bce_2, [b,c,e], ok).
visited(q_bce_3, [b,c,e], no).
visited(q_bce_4, [b,c,e], no).
visited(q_bce_5, [b,c,d,e], no).
visited(q_bce_6, [b,c,e], no).
visited(q_bce_7, [b,c,e], no).
visited(q_bce_8, [b,c,e], no).
visited(q_bce_9, [b,c,d,e], no).
visited(q_bce_10, [b,c,d,e], no).
visited(q_bce_11, [b,c,e], no).
visited(q_cde_1, [c,d,e], ok).
visited(q_cde_2, [c,d,e], ok).
visited(q_cde_3, [c,d,e], no).
visited(q_cdf_1, [c,d,f], ok).
visited(q_cdf_2, [c,d,f], ok).
visited(q_cdf_3, [c,d,f], no).
visited(q_cdf_4, [c,d,f], no).
visited(q_cdf_5, [c,d,f], no).
visited(q_abce_1, [a,b,c,e], ok).
visited(q_abce_2, [a,b,c,e], ok).
visited(q_abce_3, [a,b,c,d,e], no).
visited(q_abce_4, [a,b,c,d,e], no).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], ok).
visited(q_acef_2, [a,c,d,e,f], ok).
visited(q_cdef_1, [c,d,e,f], ok).
visited(q_cdef_2, [c,d,e,f], ok).
visited(q_bcde_1, [b,c,d,e], ok).
visited(q_bcde_2, [b,c,d,e], ok).
visited(q_bcde_3, [b,c,d,e], no).
visited(q_bcde_4, [b,c,d,e], no).
visited(q_bcde_5, [b,c,d,e], no).
visited(q_bcde_6, [b,c,d,e], no).
visited(q_bcde_7, [b,c,d,e], no).
visited(q_bcde_8, [b,c,d,e], no).
visited(q_bcde_9, [b,c,d,e], no).
visited(q_bcdf_1, [b,c,d,f], ok).
visited(q_bcdf_2, [b,c,d,f], ok).
visited(q_acde_1, [a,c,d,e], ok).
visited(q_acde_2, [a,c,d,e], ok).
visited(q_abcde_1, [a,b,c,d,e], ok).
visited(q_abcde_2, [a,b,c,d,e], ok).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,12),(b,15),(c,34),(d,20),(e,22),(f,8)]).
```

% ST ADAPTIVE TEST output st_5a

Student = [[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

```
visited(q_c_1, [c], ok).
visited(q_c_2, [c], no).
visited(q_bc_1, [b,c], ok).
visited(q_bc_2, [b,c], ok).
visited(q_bc_3, [b,c,d], ok).
visited(q_cd_1, [c,d], no).
visited(q_cd_2, [c,d], ok).
visited(q_cd_3, [c,d], ok).
visited(q_cd_4, [c,d], ok).
visited(q_cd_5, [c,d], ok).
visited(q_cd_6, [c,d], ok).
visited(q_ce_1, [c,e], no).
visited(q_ce_2, [c,e], ok).
visited(q_ce_3, [c,e], ok).
visited(q_ce_4, [c,e], ok).
visited(q_ce_5, [c,e], ok).
visited(q_ce_6, [c,e], ok).
visited(q_ace_1, [a,c,e], no).
visited(q_ace_2, [a,c,e], ok).
visited(q_ace_3, [a,c,e], ok).
visited(q_bcd_1, [b,c,d], ok).
visited(q_bcd_2, [b,c,d], ok).
visited(q_bcd_3, [b,c,d], ok).
visited(q_bcd_4, [b,c,d], ok).
visited(q_bcd_5, [b,c,d], ok).
visited(q_bce_1, [b,c,e], no).
visited(q_bce_2, [b,c,e], ok).
visited(q_bce_3, [b,c,e], ok).
visited(q_bce_4, [b,c,e], ok).
visited(q_bce_5, [b,c,d,e], ok).
visited(q_bce_6, [b,c,e], ok).
visited(q_bce_7, [b,c,e], ok).
visited(q_bce_8, [b,c,e], ok).
visited(q_bce_9, [b,c,d,e], ok).
visited(q_bce_10, [b,c,d,e], ok).
visited(q_bce_11, [b,c,e], ok).
visited(q_cde_1, [c,d,e], ok).
visited(q_cde_2, [c,d,e], ok).
visited(q_cde_3, [c,d,e], ok).
visited(q_cdf_1, [c,d,f], ok).
visited(q_cdf_2, [c,d,f], ok).
visited(q_cdf_3, [c,d,f], ok).
visited(q_cdf_4, [c,d,f], ok).
visited(q_cdf_5, [c,d,f], ok).
visited(q_abce_1, [a,b,c,e], ok).
visited(q_abce_2, [a,b,c,e], ok).
visited(q_abce_3, [a,b,c,d,e], ok).
visited(q_abce_4, [a,b,c,d,e], ok).
visited(q_abce_5, [a,b,c,e], ok).
visited(q_acef_1, [a,c,d,e,f], ok).
visited(q_acef_2, [a,c,d,e,f], ok).
visited(q_cdef_1, [c,d,e,f], ok).
visited(q_cdef_2, [c,d,e,f], ok).
visited(q_bcde_1, [b,c,d,e], ok).
visited(q_bcde_2, [b,c,d,e], ok).
visited(q_bcde_3, [b,c,d,e], ok).
visited(q_bcde_4, [b,c,d,e], ok).
visited(q_bcde_5, [b,c,d,e], ok).
visited(q_bcde_6, [b,c,d,e], ok).
visited(q_bcde_7, [b,c,d,e], ok).
visited(q_bcde_8, [b,c,d,e], ok).
visited(q_bcde_9, [b,c,d,e], ok).
visited(q_bcdf_1, [b,c,d,f], ok).
visited(q_bcdf_2, [b,c,d,f], ok).
visited(q_acde_1, [a,c,d,e], ok).
visited(q_acde_2, [a,c,d,e], ok).
visited(q_abcde_1, [a,b,c,d,e], ok).
visited(q_abcde_2, [a,b,c,d,e], ok).
```

% Summary - ST Sequential Test output

```
problems_presented(68,68).
opportunities_presented([(a,14),(b,37),(c,68),(d,44),(e,45),(f,11)]).
opportunities_correctly_applied([(a,13),(b,36),(c,63),(d,43),(e,42),(f,11)]).
```

Appendix L. Running XP1 Adaptive Test

This appendix contains the results from running XP1 for a selected list of simulated students, as described in Section 6.8. The following legend to represent the different skills:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole

% XP1 ADAPTIVE TEST output xp1_1a

Student = [[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
```

Selected Node : 5

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP1 ADAPTIVE TEST output xp1_1a

```
problems_presented(14,68).
opportunities_presented([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
opportunities_correctly_applied([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
```

% XP1 ADAPTIVE TEST output xp1_2e

Student = [[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
```

Selected Node : 3

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
```

% Summary - XP1 ADAPTIVE TEST output xp1_2e

```
problems_presented(22,68).
opportunities_presented([(a,8),(b,10),(c,22),(d,14),(e,16),(f,8)]).
opportunities_correctly_applied([(a,6),(b,8),(c,14),(d,8),(e,12),(f,0)]).
```

% XP1 ADAPTIVE TEST output xp1_2f

Student = [[sam2f-[[makeCommon],[checkAndAdd],[]]]]

```
Selected Node : 4
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).


Selected Node : 3
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).

Selected Node : 2
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), wrong).
```

Selected Node : 1

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), wrong).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), wrong).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
```

% Summary - XP1 ADAPTIVE TEST output xp1_2f

```
problems_presented(30,68).
opportunities_presented([(a,8),(b,12),(c,30),(d,16),(e,18),(f,8)]).
opportunities_correctly_applied([(a,2),(b,2),(c,6),(d,0),(e,2),(f,0)]).
```

% XP1 ADAPTIVE TEST output xp1_3a

Student = [[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
```

Selected Node : 3

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
```

Selected Node : 2

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), wrong).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), wrong).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), wrong).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), wrong).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), wrong).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), wrong).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), wrong).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), wrong).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), wrong).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), wrong).
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), correct).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), correct).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), correct).
```

% Summary - XP1 ADAPTIVE TEST output xp1_3a

```
problems_presented(28,68).
opportunities_presented([(a,8),(b,12),(c,28),(d,16),(e,18),(f,8)]).
opportunities_correctly_applied([(a,4),(b,6),(c,11),(d,1),(e,8),(f,0)]).
```

% XP1 ADAPTIVE TEST output xp1_4a

Student = [[sam4a-[[makeCommon],[checkAndAdd],[]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
```

Selected Node : 5

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP1 ADAPTIVE TEST output xp1_4a

```
problems_presented(14,68).
opportunities_presented([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
opportunities_correctly_applied([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
```

% XP1 ADAPTIVE TEST output xp1_5a

Student = [[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 4

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
```

Selected Node : 5

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(4, [a,b,c,e], q_abce_1, fr(1:2/3,1:3/5), correct).
visited(4, [a,b,c,e], q_abce_2, fr(2:1/2,7/3), correct).
visited(4, [a,c,e,f], q_acef_1, fr(1:1/3,2/3), correct).
visited(4, [a,c,e,f], q_acef_2, fr(1:1/5,4/5), correct).
visited(4, [c,d,e,f], q_cdef_1, fr(1/3,5/3), correct).
visited(4, [c,d,e,f], q_cdef_2, fr(3/2,5/2), correct).
visited(4, [b,c,d,e], q_bcde_1, fr(3/8,3/4), correct).
visited(4, [b,c,d,e], q_bcde_2, fr(2/3,5/6), correct).
visited(4, [b,c,d,f], q_bcdf_1, fr(3/6,2/4), correct).
visited(4, [b,c,d,f], q_bcdf_2, fr(5/10,4/8), correct).
visited(4, [a,c,d,e], q_acde_1, fr(1:1/8,1:3/8), correct).
visited(4, [a,c,d,e], q_acde_2, fr(1:1/6,2:1/6), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP1 ADAPTIVE TEST output xp1_5a

```
problems_presented(14,68).
opportunities_presented([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
opportunities_correctly_applied([(a,8),(b,8),(c,14),(d,10),(e,12),(f,6)]).
```

Appendix M. Running XP2 Adaptive Test

This appendix contains the results from running XP2 for a selected list of simulated students, as described in Section 6.8. The following legend to represent the different skills:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole

```
% XP2 ADAPTIVE TEST output xp2_1a

Student = [[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 3
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).

Selected Node : 5
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).

Selected Node : 6
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).

% Summary - XP2 ADAPTIVE TEST output
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
```

% XP2 ADAPTIVE TEST output xp2_2e

Student = [[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]

```
Selected Node : 3
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).

Selected Node : 5
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).

Selected Node : 6
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).

% Summary - XP2 ADAPTIVE TEST output
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,10),(d,6),(e,8),(f,0)]).
```

% XP2 ADAPTIVE TEST output xp2_2f

Student = [[sam2f-[[makeCommon],[checkAndAdd],[]]]]

```
Selected Node : 3
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).

Selected Node : 1
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).

Selected Node : 2
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), wrong).
```

% Summary - XP2 ADAPTIVE TEST output

```
problems_presented(18,68).
opportunities_presented([(a,2),(b,6),(c,18),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,2),(b,2),(c,6),(d,0),(e,2),(f,0)]).
```

% XP2 ADAPTIVE TEST output xp2_3a

Student = [[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
```

Selected Node : 1

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
```

Selected Node : 2

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), correct).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), correct).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), correct).
```

% Summary - XP2 ADAPTIVE TEST output

```
problems_presented(18,68).
opportunities_presented([(a,2),(b,6),(c,18),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,2),(b,4),(c,11),(d,1),(e,6),(f,0)]).
```

% XP2 ADAPTIVE TEST output xp2_4a

Student = [[sam4a-[[makeCommon],[checkAndAdd],[]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP2 ADAPTIVE TEST output

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
```

% XP2 ADAPTIVE TEST output xp2_5a

Student = [[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP2 ADAPTIVE TEST output

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,3),(b,5),(c,10),(d,8),(e,6),(f,2)]).
```

Appendix N. Running XP3 Adaptive Test

This appendix contains the results from running XP3 for a selected list of simulated students, as described in Section 6.8. The following legend to represent the different skills:

a	makeVulgar
b	makeCommon
c	checkAndAdd
d	cancel
e	makeProper
f	makeWhole

% XP3 ADAPTIVE TEST output xp3_1a

Student = [[sam1a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP3 ADAPTIVE TEST output xp3_1a

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
```

% XP3 ADAPTIVE TEST output xp3_2e

Student = [[sam2e-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP3 ADAPTIVE TEST output xp3_2e

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,10),(d,6),(e,8),(f,0)]).
```

% XP3 ADAPTIVE TEST output xp3_2f

Student = [[sam2f-[[makeCommon],[checkAndAdd],[]]]]

```
Selected Node : 3
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).

Selected Node : 1
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).

Selected Node : 2
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), wrong).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), wrong).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), wrong).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), wrong).
```

% Summary - XP3 ADAPTIVE TEST output xp3_2f

```
problems_presented(18,68).
opportunities_presented([(a,2),(b,6),(c,18),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,2),(b,2),(c,6),(d,0),(e,2),(f,0)]).
```

% XP3 ADAPTIVE TEST output xp3_3a

Student = [[sam3a-[[makeVulgar,makeCommon],[checkAndAdd],[malCancel,makeProper,makeWhole]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
```

Selected Node : 1

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
```

Selected Node : 2

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), wrong).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), wrong).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), wrong).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), wrong).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), wrong).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), wrong).
visited(1, [c], q_c_1, fr(1/3,1/3), correct).
visited(1, [c], q_c_2, fr(3/5,1/5), correct).
visited(2, [b,c], q_bc_1, fr(1/2,1/5), correct).
visited(2, [b,c], q_bc_2, fr(1/3,1/5), correct).
visited(2, [c,d], q_cd_1, fr(4/9,2/9), wrong).
visited(2, [c,d], q_cd_2, fr(12/64,4/64), correct).
visited(2, [c,e], q_ce_1, fr(5/7,6/7), correct).
visited(2, [c,e], q_ce_2, fr(4/7,8/7), correct).
```

% Summary - XP3 ADAPTIVE TEST output xp3_3a

```
problems_presented(18,68).
opportunities_presented([(a,2),(b,6),(c,18),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,2),(b,4),(c,11),(d,1),(e,6),(f,0)]).
```

% XP3 ADAPTIVE TEST output xp3_4a

Student = [[sam4a-[[makeCommon],[checkAndAdd],[]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), correct).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), correct).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP3 ADAPTIVE TEST output xp3_4a

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
```

% XP3 ADAPTIVE TEST output xp3_5a

Student = [[sam5a-[[makeVulgar,makeCommon],[checkAndAdd],[cancel,makeProper,makeWhole]]]]

Selected Node : 3

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
```

Selected Node : 5

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

Selected Node : 6

```
Visited list :
visited(3, [a,c,e], q_ace_1, fr(1:1/5,2/5), wrong).
visited(3, [a,c,e], q_ace_2, fr(2:3/7,2/7), correct).
visited(3, [b,c,d], q_bcd_1, fr(1/2,1/4), correct).
visited(3, [b,c,d], q_bcd_2, fr(5/6,1/8), correct).
visited(3, [b,c,e], q_bce_1, fr(2/3,3/5), wrong).
visited(3, [b,c,e], q_bce_2, fr(1/2,7/3), correct).
visited(3, [c,d,e], q_cde_1, fr(5/8,9/8), correct).
visited(3, [c,d,e], q_cde_2, fr(8/9,4/9), correct).
visited(3, [c,d,f], q_cdf_1, fr(5/7,2/7), correct).
visited(3, [c,d,f], q_cdf_2, fr(4/5,1/5), correct).
visited(5, [a,b,c,d,e], q_abcde_1, fr(1/3,1:5/6), correct).
visited(5, [a,b,c,d,e], q_abcde_2, fr(1/4,2:1/12), correct).
```

% Summary - XP3 ADAPTIVE TEST output xp3_5a

```
problems_presented(12,68).
opportunities_presented([(a,4),(b,6),(c,12),(d,8),(e,8),(f,2)]).
opportunities_correctly_applied([(a,3),(b,5),(c,10),(d,8),(e,6),(f,2)]).
```

Appendix O. Tabulated Results of Different Students

This appendix contains the tabulated results of running different assessors for a selected list of simulated students, as discussed in Chapter 9. The assessors are XP, ST, XP1, XP2 and XP3. The table identifier is given at the top left hand corner of each table. For example ‘XP_1a’ means the tabulated results of running XP for simulated student *sam1a*. The shaded skills are the skills mastered by the student and instantiated during the creation of each simulated student (see Section 6.3).

Table XP _1a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	4	4	0	1.00	-
c. checkAndAdd	68	7	7	0	1.00	-
d. cancel	44	5	5	0	1.00	-
e. makeProper	45	6	6	0	1.00	-
f. makeWhole	11	3	3	0	1.00	-
Total:	219	29	29	0		
Average:					1.00	-

Table XP_2e

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	3	1	0.75	-
b. makeCommon	37	5	4	1	0.80	-
c. checkAndAdd	68	11	7	4	0.64	-
d. cancel	44	7	4	3	0.57	-
e. makeProper	45	8	6	2	0.75	-
f. makeWhole	11	4	0	4	-	1.00
Total:	219	39	24	15		
Average:					0.70	1.00

Table XP_2f

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	1	3	-	0.75
b. makeCommon	37	6	1	5	0.17	-
c. checkAndAdd	68	15	3	12	0.20	-
d. cancel	44	8	0	8	-	1.00
e. makeProper	45	9	1	8	-	0.89
f. makeWhole	11	4	0	4	-	1.00
Total:	219	46	6	40		
Average:					0.18	0.91

Table XP_3a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	2	2	0.50	-
b. makeCommon	37	6	3	3	0.50	-
c. checkAndAdd	68	14	5	9	0.36	-
d. cancel	44	8	0	8	-	1.00
e. makeProper	45	9	4	5	0.44	-
f. makeWhole	11	4	0	4	0.00	-
Total:	219	45	14	31		
Average:					0.36	1.00

Table XP_4a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	-	0.00
b. makeCommon	37	4	4	0	1.00	-
c. checkAndAdd	68	7	7	0	1.00	-
d. cancel	44	5	5	0	-	0.00
e. makeProper	45	6	6	0	-	0.00
f. makeWhole	11	3	3	0	-	0.00
Total:	219	29	29	0		
Average:					1.00	0.00

Table XP_5a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	4	4	0	1.00	-
c. checkAndAdd	68	7	7	0	1.00	-
d. cancel	44	5	5	0	1.00	-
e. makeProper	45	6	6	0	1.00	-
f. makeWhole	11	3	3	0	1.00	-
Total:	219	29	29	0		
Average:					1.00	-

Table XP1_1a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	8	0	1.00	-
b. makeCommon	37	8	8	0	1.00	-
c. checkAndAdd	68	14	14	0	1.00	-
d. cancel	44	10	10	0	1.00	-
e. makeProper	45	12	12	0	1.00	-
f. makeWhole	11	6	6	0	1.00	-
Total:	219	58	58	0		
Average:					1.00	-

Table XP2_2e

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	6	2	0.75	-
b. makeCommon	37	10	8	2	0.80	-
c. checkAndAdd	68	22	14	8	0.64	-
d. cancel	44	14	8	6	0.57	-
e. makeProper	45	16	12	4	0.75	-
f. makeWhole	11	8	0	8	-	1.00
Total:	219	78	48	30		
Average:					0.70	1.00

Table XP1_2f

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	2	6	-	0.75
b. makeCommon	37	12	2	10	0.17	-
c. checkAndAdd	68	30	6	24	0.20	-
d. cancel	44	16	0	16	-	1.00
e. makeProper	45	18	2	16	-	0.89
f. makeWhole	11	8	0	8	-	1.00
Total:	219	92	12	80		
Average:					0.18	0.91

Table XP1_3a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	4	4	0.50	-
b. makeCommon	37	12	6	6	0.50	-
c. checkAndAdd	68	28	11	17	0.39	-
d. cancel	44	16	1	15	-	0.94
e. makeProper	45	18	8	10	0.44	-
f. makeWhole	11	8	0	8	0.00	-
Total:	219	90	30	60		
Average:					0.37	0.94

Table XP1_4a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	8	0	-	0.00
b. makeCommon	37	8	8	0	1.00	-
c. checkAndAdd	68	14	14	0	1.00	-
d. cancel	44	10	10	0	-	0.00
e. makeProper	45	12	12	0	-	0.00
f. makeWhole	11	6	6	0	-	0.00
Total:	219	58	58	0		
Average:					1.00	0.00

Table XP1_5a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	8	8	0	1.00	-
b. makeCommon	37	8	8	0	1.00	-
c. checkAndAdd	68	14	14	0	1.00	-
d. cancel	44	10	10	0	1.00	-
e. makeProper	45	12	12	0	1.00	-
f. makeWhole	11	6	6	0	1.00	-
Total:	219	58	58	0		
Average:					1.00	-

Table XP2_1a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	12	0	1.00	-
d. cancel	44	8	8	0	1.00	-
e. makeProper	45	8	8	0	1.00	-
f. makeWhole	11	2	2	0	1.00	-
Total:	219	40	40	0		
Average:					1.00	-

Table XP2_2e

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	10	2	0.83	-
d. cancel	44	8	6	2	0.75	-
e. makeProper	45	8	8	0	1.00	-
f. makeWhole	11	2	0	2	-	1.00
Total:	219	40	34	6		
Average:					0.92	1.00

Table XP2_2f

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	2	2	0	-	0.00
b. makeCommon	37	6	2	4	0.33	-
c. checkAndAdd	68	18	6	12	0.33	-
d. cancel	44	8	0	8	-	1.00
e. makeProper	45	8	2	6	-	0.75
f. makeWhole	11	2	0	2	-	1.00
Total:	219	44	12	32		
Average:					0.33	0.69

Table XP2_3a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	2	2	0	1.00	-
b. makeCommon	37	6	4	2	0.67	-
c. checkAndAdd	68	18	11	7	0.61	-
d. cancel	44	8	1	7	-	0.88
e. makeProper	45	8	6	2	0.75	-
f. makeWhole	11	2	0	2	0.00	-
Total:	219	44	24	20		
Average:					0.61	0.88

Table XP2_4a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	-	0.00
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	12	0	1.00	-
d. cancel	44	8	8	0	-	0.00
e. makeProper	45	8	8	0	-	0.00
f. makeWhole	11	2	2	0	-	0.00
Total:	219	40	40	0		
Average:					1.00	0.00

Table XP2_5a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	3	1	0.75	-
b. makeCommon	37	6	5	1	0.83	-
c. checkAndAdd	68	12	10	2	0.83	-
d. cancel	44	8	8	0	1.00	-
e. makeProper	45	8	6	2	0.75	-
f. makeWhole	11	2	2	0	1.00	-
Total:	219	40	34	6		
Average:					0.86	-

Table XP3_1a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	12	0	1.00	-
d. cancel	44	8	8	0	1.00	-
e. makeProper	45	8	8	0	1.00	-
f. makeWhole	11	2	2	0	1.00	-
Total:	219	40	40	0		
Average:					1.00	-

Table XP3_2e

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	1.00	-
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	10	2	0.83	-
d. cancel	44	8	6	2	0.75	-
e. makeProper	45	8	8	0	1.00	-
f. makeWhole	11	2	0	2	-	1.00
Total:	219	40	34	6		
Average:					0.92	1.00

Table XP3_2f

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	2	2	0	-	0.00
b. makeCommon	37	6	2	4	0.33	-
c. checkAndAdd	68	18	6	12	0.33	-
d. cancel	44	8	0	8	-	1.00
e. makeProper	45	8	2	6	-	0.75
f. makeWhole	11	2	0	2	-	1.00
Total:	219	44	12	32		
Average:					0.33	0.69

Table XP3_3a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	2	2	0	1.00	-
b. makeCommon	37	6	4	2	0.67	-
c. checkAndAdd	68	18	11	7	0.61	-
d. cancel	44	8	1	7	-	0.88
e. makeProper	45	8	6	2	0.75	-
f. makeWhole	11	2	0	2	0.00	-
Total:	219	44	24	20		
Average:					0.61	0.88

Table XP3_4a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	4	0	-	0.00
b. makeCommon	37	6	6	0	1.00	-
c. checkAndAdd	68	12	12	0	1.00	-
d. cancel	44	8	8	0	-	0.00
e. makeProper	45	8	8	0	-	0.00
f. makeWhole	11	2	2	0	-	0.00
Total:	219	40	40	0		
Average:					1.00	0.00

Table XP3_5a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	4	3	1	0.75	-
b. makeCommon	37	6	5	1	0.83	-
c. checkAndAdd	68	12	10	2	0.83	-
d. cancel	44	8	8	0	1.00	-
e. makeProper	45	8	6	2	0.75	-
f. makeWhole	11	2	2	0	1.00	-
Total:	219	40	34	6		
Average:					0.86	-

Table ST_1a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	14	0	1.00	-
b. makeCommon	37	37	37	0	1.00	-
c. checkAndAdd	68	68	68	0	1.00	-
d. cancel	44	44	44	0	1.00	-
e. makeProper	45	45	45	0	1.00	-
f. makeWhole	11	11	11	0	1.00	-
Total:	219	219	219	0		
Average:					1.00	-

Table ST_2e

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	12	2	0.86	-
b. makeCommon	37	37	35	2	0.95	-
c. checkAndAdd	68	68	57	11	0.84	-
d. cancel	44	44	33	11	0.75	-
e. makeProper	45	45	41	4	0.91	-
f. makeWhole	11	11	0	11	-	1.00
Total:	219	219	178	41		
Average:					0.86	1.00

Table ST_2f

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	4	10	-	0.71
b. makeCommon	37	37	3	34	0.08	-
c. checkAndAdd	68	68	8	60	0.12	-
d. cancel	44	44	0	44	-	1.00
e. makeProper	45	45	4	41	-	0.91
f. makeWhole	11	11	0	11	-	1.00
Total:	219	219	19	200		
Average:					0.10	0.91

Table ST_3a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	6	8	0.43	-
b. makeCommon	37	37	13	24	0.35	-
c. checkAndAdd	68	68	25	43	0.37	-
d. cancel	44	44	1	43	-	0.98
e. makeProper	45	45	20	25	0.44	-
f. makeWhole	11	11	0	11	0.00	-
Total:	219	219	65	154		
Average:					0.32	0.98

Table ST_4a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	12	2	-	0.14
b. makeCommon	37	37	15	22	0.41	-
c. checkAndAdd	68	68	34	34	0.50	-
d. cancel	44	44	20	24	-	0.55
e. makeProper	45	45	22	23	-	0.51
f. makeWhole	11	11	8	3	-	0.27
Total:	219	219	111	108		
Average:					0.45	0.37

Table ST_5a

Skills	Total no.of opportunities	No. of opportunities presented	No. of opportunities correctly applied	No. of opportunities wrongly applied	Accuracy on mastered skill	Accuracy on unmastered skill
a. makeVulgar	14	14	13	1	0.93	-
b. makeCommon	37	37	36	1	0.97	-
c. checkAndAdd	68	68	63	5	0.93	-
d. cancel	44	44	43	1	0.98	-
e. makeProper	45	45	42	3	0.93	-
f. makeWhole	11	11	11	0	1.00	-
Total:	219	219	208	11		
Average:					0.96	-

Appendix P. Summary of Performance of Assessors

A summary of results from running all five assessors – XP, ST, XP1, XP2 and XP3 – for different types of simulated students is given in the following table.

		Skills Presented			Skills Correctly Assessed			Accuracy in Assessment		
		Mastered	Unmastered	Total	Mastered	Unmastered	Total	Mastered	Unmastered	Overall
Sam1	**ST**	1752	0	1752	1506	-	1506	0.86	-	0.86
	XP	292	0	292	202	-	202	0.69	-	0.69
	XP1	584	0	584	404	-	404	0.69	-	0.69
	XP2	320	0	320	284	-	284	0.89	-	0.89
	XP3	320	0	320	284	-	284	0.89	-	0.89
Sam2	**ST**	1049	265	1314	573	245	818	0.55	0.92	0.62
	XP	188	55	243	99	50	149	0.53	0.91	0.61
	XP1	376	110	486	198	100	298	0.53	0.91	0.61
	XP2	246	54	300	164	46	210	0.67	0.85	0.70
	XP3	208	48	256	150	40	190	0.72	0.83	0.74
Sam3	**ST**	735	360	1095	172	297	469	0.23	0.83	0.43
	XP	147	68	215	44	56	100	0.30	0.82	0.47
	XP1	294	136	430	89	111	200	0.30	0.82	0.47
	XP2	198	86	284	83	61	144	0.42	0.71	0.51
	XP3	122	74	196	55	49	104	0.45	0.66	0.53
Sam4	**ST**	672	423	1095	271	300	571	0.40	0.71	0.52
	XP	95	70	165	74	17	91	0.78	0.24	0.55
	XP1	210	154	364	121	73	194	0.58	0.47	0.53
	XP2	226	122	348	137	53	190	0.61	0.43	0.55
	XP3	140	76	216	104	28	132	0.74	0.37	0.61
Sam5	**ST**	862	14	876	738	4	742	0.86	0.29	0.85
	XP	138	4	142	76	2	78	0.55	0.50	0.55
	XP1	244	8	252	169	4	173	0.69	0.50	0.69
	XP2	246	10	256	182	4	186	0.74	0.40	0.73
	XP3	202	2	204	158	0	158	0.78	0.00	0.77

Table 9. Comparing Five Assessors for Five types of Simulated Students

Appendix Q. List of Publications

This appendix contains a list of publications of the author.

a) *Premodelling for Examination Revision through Adaptive Testing.* Chua Abdullah, S. In Proceedings of the 3rd Human Centred Technology Postgraduate Workshop “Interacting through/with Technology: Increasing the Potential for Communicating and Learning?”, HCT'99, 30th September - 1st October 1999, Brighton, UK, organised by the University of Sussex, School of Cognitive and Computing Sciences.

b) *Using Constraints to Develop and Deliver Adaptive Tests.* Chua Abdullah, S. and Cooley, R.E. In H. Cooper and S. Clowes, editors, Proceedings of the Fourth International Computer Assisted Assessment Conference, Loughborough University, UK, pages 93-101, June 2000.

c) *Modelling Human Testing Strategies: A Computer-Aided Approach to Knowledge Acquisition.* Chua Abdullah, S. and Cooley, R.E. In Proceedings of Workshop W1 on Modeling Human Teaching Tactics and Strategies, held as part of the Fifth International Conference on Intelligent Tutoring Systems, ITS'2000, Montréal, Canada, page 17, June 2000.

d) *The Use of Constraint Logic Programming in the Development of Adaptive Tests.* Chua Abdullah, S. and Cooley, R.E. In G. Gauthier, C. Frasson and K. VanLehn, editors, Lecture Notes in Computer Science 1839, Proceedings of the Fifth International Conference on Intelligent Tutoring Systems, ITS 2000, Montréal, Canada, page 650. Springer-Verlag, June 2000.

e) *Controlling Problem Progression in Adaptive Testing*. Cooley, R.E. and Chua Abdullah, S. Proceedings of the International Conference on Computers in Education and the International Conference on Computer Assisted Instruction, ICCE/ICCAI 2000, Taiwan, November 2000.

f) *Using Simulated Students to Evaluate an Adaptive Testing System.* Chua Abdullah, S. and Cooley, R.E. (2002). Proceedings of the *International Conference on Computers in Education, ICCE 2002*, New Zealand, December 2002.

Bibliography

Abbott,J. (1995). A Matter of Constraint. Unix News (82), pp.18-19.

Anderson,J.R. (1983). The Architecture of Cognition. Harvard University Press, Cambridge, Massachusetts, London.

Anderson,J.R. (1992). Intelligent Tutoring and High School Mathematics. In: Frasson,C., Gauthier,G., McCalla,G.I., (Eds.), Lecture Notes in Computer Science 608. Intelligent Tutoring Systems. Second International Conference, ITS'92, Montreal, Canada, June 1992 Proceedings. Springer-Verlag, Berlin Heidelberg New York.

Anderson,J.R. (1998a). Applications and Misapplications of Cognitive Psychology to Mathematics Education. http://act.psy.cmu.edu/personal/ja/misapplied.html

Anderson,J.R. (1998b). Instructional Applications of ACT: Past, Present, and Future. In: Goettl,B.P., Halff,H.M., Redfield,C.L., Shute,V., (Eds.), Intelligent Tutoring Systems: Proceedings of 4th International Conference, ITS'98, San Antonio, Texas, USA, August 1998, No.1452 Lectures Notes in Computer Science. Springer, Berlin, Heidelberg, p.1.

Anderson,J.R. (1988c). The Expert Module. In: Polson,M.C., Richardson,J.J., (Eds.), Foundations of Intelligent Tutoring Systems. Lawrence Erlbaum Associates, Hillsdale, New Jersey, Hove and London, pp. 21-53.

Anderson,J.R., Boyle,C.F., Corbett,A.T., Lewis,M.W. (1990). Cognitive Modeling and Intelligent Tutoring. Artificial Intelligence 42 (1), pp. 7-49.

Anderson,J.R., Boyle,C.F., Yost,G. (1985). The Geometry Tutor. In: Morgan Kaufmann, (Ed.), Los Angeles, pp. 1-7.

Anderson,J.R., Corbett,A.T., Koedinger,K., Pelletier,R. (1995. Cognitive tutors: Lessons Learned. The Journal of Learning Sciences 4, pp.167-207.

Anderson, J.R., Reiser, B.J. (1985). The LISP Tutor. BYTE April 1985, pp. 159-175.

Arroyo, I., Conejo, R., Guzman, E., Woolf, B.P. (2001) An adaptive web-based component for cognitive ability estimation. Proceedings of the Tenth International Conference on Artificial Intelligence in Education. San Antonio, Texas, May 2001. pp. 456-466, IOS Press.

Baffes,P., Mooney,R. (1996). Refinement-Based Student Modeling and Automated Bug Library Construction. Journal of Artificial Intelligence in Education 7 (1), pp.75-117.

Barr,A., Feigenbaum,E.A. (1981). The Handbook of Artificial Intelligence, Volume 2. Pitman, London.

Baumunk,K., Dowling,C.K. (1997). Validity of Spaces for Assessing Knowledge about Fractions. Journal of Mathematical Psychology 41, pp.99-105.

Bayes, Rev. T. (1753). An Essay toward solving a Problem in the Doctrine of Chances, Philos. Trans. R. Soc. London, 53, pp.370-418, reprinted in Biometrika, 45, 1958, pp.293-315.

Beck,J., Stern,M., Haugsjaa,E. (1998). Applications of AI in Education. ACM CrossRoads. http://www.acm.org/crossroads/xrds3-1/aied.html

Beck,J., Stern,M., Woolf,B.P. (1997). Using the Student Model to Control Problem Difficulty. In: Jameson,A., Paris,C., Tasso,C., (Eds.), CISM Courses and Lectures no.383. International Centre for Mechanical Sciences. User Modeling. Proceedings of the Sixth International Conference UM97. SpringerWien, New York, pp. 277-289.

Bloom,B.S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. Educational Researcher 13 (6), pp.4-16.

Borasi,R. (1994). Capitalizing on Errors as "Springboards for Inquiry": A Teaching Experiment. Journal for Research in Mathematics Education 25 (2), pp. 166-208.

Boy,G. (1996). Learning Evolution and Software Agents Emergence. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Intelligent Tutoring Systems. Third International Conference, ITS'96, Montreal, Canada, June 1996 Proceedings. Lecture Notes in Computer Science 1086. Springer-Verlag, Berlin Heidelberg, pp. 10-25.

Brown,J.S., Burton,R.R. (1978). Diagnostic Models for Procedural Bugs in Basic Mathematical Skills. Cognitive Science 2, pp.155-192.

Brown,J.S., Burton,R.R., Bell,A.G. (1975). SOPHIE: A Step Toward Creating a Reactive Learning Environment. International Journal Man-Machine Studies 7, pp. 675-696.

Brown,J.S., VanLehn,K. (1980). Repair Theory: A Generative Theory of Bugs in Procedural Skills. Cognitive Science 4, pp. 379-426.

Brusilovskiy,P.L. (1994). The Construction and Application of Student Models in Intelligent Tutoring Systems. Journal of Computer and Systems Sciences International 32 (1), pp.70-89.

Bull,S., Shurville,S. (1999). Cooperative Writer Modelling: Facilitating Reader-Based Writing with SCRAWL. pp. 1-9.

Burton,R.R., Brown,J.S. (1985). An Investigation of Computer Coaching for Informal Learning Activities. International Journal Man-Machine Studies 23 (1).

Carbonell,J.R. (1970). AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction. IEEE Transactions on Man-Machine Systems MMS-11, December 1970, 4.

Carlsson,M., Ottosson,G., Carlson,B. (1997). An Open-Ended Finite Constraint Solver. In: Proc. Programming Languages: Implementations, Logic and Programs.

Chan,T.W. (1992). Curriculum Tree - A Knowledge-Based Architecture for Intelligent Tutoring Systems. Lecture Notes in Computer Science 608, pp.140-147.

Chan, T.W. and Baskin, A.B. (1990). Learning companion systems. In: Frasson, C. & Gauthier, G., (Eds.), Intelligent Tutoring Systems: At the crossroads of Artificial Intelligence and Education, Ablex, N.J.

Chan, T.W. and Chou, C.Y. (1995). Simulating a Learning Companion in Reciprocal Tutoring Systems. Proceedings of Computer Support for Collaborative Learning '95, Indiana University, Bloomington, IN.

Charniak,E. (1991). Bayesian Networks Without Tears.. AI Magazine 12 (4), Winter 91, pp.50-63.

Chi,M.T.H., Bassok,M., Lewis,M.W., Reimann,P., Glaser,R. (1989). Self-Explanations - How Students Study and Use Examples in Learning to Solve Problems. Cognitive Science 13 (2), pp.145-182.

Chi,M.T.H., Feltovich,P.J., Glaser,R. (1981). Categorization and Representation of Physics Problems by Experts and Novices. Cognitive Science 5 (2), pp.121-152.

Clancey,W.J. (1979). Tutoring Rules for Guiding a Case Method Dialogue. International Journal Man-Machine Studies 11, pp.25-49.

Clancey,W.J. (1986). Qualitative Student Models. Annual Review of Computer Science 1, pp.381-450.

Clancey,W.J. (1987). Knowledge-Based Tutoring: The GUIDON Program. MIT Press, Cambridge, MA.

Clancey,W.J., Soloway,E. (1990). Artificial Intelligence and Learning Environment. Artificial Intelligence 41, pp.1-6.

Cohen,J. (1990). Constraint Logic Programming Languages. Communications of the ACM 33 (7), pp.52-68.

Cohen,P. (1996). Logic Programming and Constraint Logic Programming. ACM Computing Surveys 28 (1), pp.257-259.

Collins,J.A., Greer,J.E., Huang,S.X. (1996). Adaptive Assessment Using Granularity Hierarchies and Bayesian Nets. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Lecture Notes in Computer Science 1086. Intelligent Tutoring Systems. Third International Conference, ITS'96, Montréal, Canada, June 1996 Proceedings. Springer-Verlag, Berlin Heidelberg, pp.569-577.

COMPASS (2000). COMPASS - Computerized Adaptive Placement Assessment and Support System. http://www.act.org/compass/index.html

Constraint Programming (2000). http://www.aiai.ed.ac.uk/links/constr.html

Cooper,B., Dunne,M. (2000). Assessing Children's Mathematical Knowledge. Open University Press, Buckingham, Philadelphia.

Covington,M.V., Omelich,C.L. (1987). "I knew it cold before the exam": A test of the anxiety-blockage hypothesis. Journal of Educational Psychology 79 (4), pp.393-400.

Darmoni,S.J.,Fajner,A.,Mahe,N.,Leforestier,A.,Vondracek,M.,Stelian,O.,Baldenweck,M.(2000). Horoplan: Computer-Assisted Nurse Scheduling using Constraint-based Programming. http://www.chu-rouen.fr/dsii/publi/plao.html

Davis,R.B. (1984). Learning Mathematics - The Cognitive Science Approach to Mathematics Education. Croom Helm, London, Sydney.

Deboys,M., Pitt,E. (1988). Lines of Development in Primary Mathematics. The Blackstaff Press, Belfast.

Desmoulins,C., van Labeke,N. (1996). Towards Student Modelling in Geometry with Inductive Logic Programming. www.loria.fr/~vanlabek/Papers/Euro-AIED96.html

Doignon,J.P., Falmagne,J.C. (1985). Spaces for the Assessment of Knowledge. International Journal of Man-Machine Studies 23 (2), pp.175-196.

Doignon,J.P., Falmagne,J.C. (1998). Knowledge Spaces. Springer-Verlag, Berlin.

Dowling,C.E. (1993). Applying the Basis of a Knowledge Space for Controlling the Questioning of an Expert. Journal of Mathematical Psychology 37, pp.21-48.

Dowling,C.E., Hockemeyer,C., Ludwig,A.H. (1996). Adaptive Assessment and Training Using the Neighbourhood of Knowledge States. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Intelligent Tutoring Systems, Third International Conference, ITS '96, Montréal, Canada, June12-14, 1996, Proceedings Lecture Notes in Computer Science, Vol.1086. Springer, Berlin, Heidelberg, pp. 578-587.

Dowling,C.E., Kaluscha,R. (1995). Prerequisite Relationships for the Adaptive Assessment of Knowledge. In: J.Greer, (Ed.), Proceedings of AI-ED'95, 7th World Conference on Artificial Intelligence in Education, Washington, DC, 16-19 August 1995, AACE. pp. 43-50.

du Boulay,B. (2000a). Workshop W1 on Modeling Human Teaching Tactics and Strategies, held as part of the Fifth International Conference on Intelligent Tutoring Systems, ITS'2000, June 19, Montréal, Canada.

du Boulay,B. (2000b). Can we learn from ITSs? In: Gauthier,G., Frasson,C., VanLehn,K., (Eds.), Intelligent Tutoring Systems: Proceedings of 5th International Conference, ITS 2000, Montreal, Canada, No.1839 in Lectures Notes in Computer Science. Springer, Berlin, Heidelberg, New York, pp.9-17.

Educational Testing Service (1999). http://www.ets.org/.

Elsom-Cook,M. (1988). Guided Discovery Tutoring and Bounded User Modelling. In: Self,J., (Ed.), Artificial Intelligence and Human Learning - Intelligent Computer-aided Instruction. Chapman and Hall, London, New York, pp. 165-178.

Falmagne,J.C., Doignon,J.P., Koppen,M., Villano,M., Johannesen,L. (1990). Introduction to Knowledge Spaces - How to Build, Test, and Search Them. Psychological Review 97 (2), pp.201-224.

Frasson,C., Mengelle,T., Aimeur,E., Gouarderes,G. (1996). An Actor-Based Architecture for Intelligent Tutoring Systems. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Intelligent Tutoring Systems. Third International Conference, ITS'96, Montreal, Canada, June 1996 Proceedings. Lecture Notes in Computer Science 1086. Springer, Berlin Heidelberg, pp. 57-65.

Frederiksen,J.R., White,B.Y. (1990). Intelligent Tutors as Intelligent Testers. In: Frederiksen,N., Glaser,R., Lesgold,A., Shafto,M.G., (Eds.), Diagnostic Monitoring of Skill and Knowledge Acquisition. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, Hove, London, pp.1-26.

Frosini,G., Lazzerini,B., Marcelloni,F. (1998). Performing automatic exams. Computers & Education 31 (3), pp.281-300.

Gemini (2000). White Paper. http://www.gemini.com/swift/whitepap.htm

Gilmore,D., Self,J. (1988). The Application of Machine Learning to Intelligent Tutoring Systems. In: Self,J., (Ed.), Artificial Intelligence and Human Learning - Intelligent Computer-aided Instruction. Chapman and Hall, London, New York, pp. 179-196.

Glass,M., Kim,J.H., Evens,M.W., Michael,J.A., Rovick,A.A. (1999). Novice vs. Expert Tutors: A Comparison of Style. Tenth Midwest Artificial Intelligence and Cognitive Science Conference (MAICS '99), Bloomington, Indiana.

GMAT (2000). http://testprep.embark.com/gmat/freeinfo/gmat_article_overview.asp

Goldstein,I.P. (1982). The Genetic Graph: A Representation for the Evolution of Procedural Knowledge. In: Sleeman,D.H., Brown,J.S., (Eds.), Intelligent Tutoring Systems. Academic Press, London.

GRE (2000). http://www.gre.org

Greer,J., McCalla,G.I. (1991). Student Modelling: The Key to Individualized Knowledge-Based Instruction, Springer-Verlag, Berlin, Heidelberg, New York.

Gugerty,L. (1997). Non-diagnostic intelligent tutoring systems: Learning collaboratively without student models. Instructional Science 25, pp.409-432.

Gutwin,C., Jones,M., Brackett,P., Massie Adolphe,K. (2000). Bringing ITS to the Marketplace: A Successful Experiment in Minimalist Design. http://www.iicm.edu/jucs_1_3/bringing_its_to_the/html/paper.htm

Halff,H.M. (1988). Curriculum and Instruction in Automated Tutors. In: Polson,M.C., Richardson,J.J., (Eds.), Foundations of Intelligent Tutoring Systems. Lawrence Erlbaum Associates, Hillsdale, New Jersey, Hove and London, pp. 79-108.

Hall, R. (2002). Thought Processes in Simplifying an Algebraic Expression, Philosophy of Mathematics Education Journal, 15, http://www.ex.ac.uk/~PErnest/pome15/processes.htm.

Hirashima,T., Kashihara,A., Toyoda,J. (1996). Toward a Learning Environment Allowing Learner-Directed Problem Practice - Helping Problem-Solving by Using Problem Simplification. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Intelligent Tutoring Systems. Third International Conference, ITS'96, Montréal, Canada, June 1996 Proceedings. Lecture Notes in Computer Science 1086. Springer, Berlin Heidelberg, pp. 466-474.

Hockemeyer,C., Dietrich,A.(1999). The adaptive tutoring RATH: a prototype. International Workshop Interactive Computer aided Learning ICL'99. Villach, Austria.

Holt,P., Dubs,S., Jones,M., Greer,J.(1994). The State of Student Modelling. In: Greer,J.E., McCalla,G.I., (Eds.), Student Modelling: The Key to Individualized Knowledge-Based Instruction. Springer-Verlag, Berlin Heidelberg, pp. 3-35.

Huang,S.X. (1996). A Content-Balanced Adaptive Testing Algorithm for Computer-Based Training Systems. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Intelligent Tutoring Systems. Third International Conference, ITS'96, Montréal, Canada, June 1996 Proceedings. Lecture Notes in Computer Science 1086. Springer, Berlin Heidelberg, pp. 306-314.

Jaffar,J., Lassez,J.-L. (1987). Constraint logic programming. Fourteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of programming languages, pp. 111-119.

Jensen,F.V. (1996). An Introduction to Bayesian Networks. UCL Press, London.

Johnson, W.L. (1990). Understanding and debugging novice programs. Artificial Intelligence, 42, pp.51-97.

Kambouri,M., Koppen,M., Villano,M., Falmagne,J.C. (1994). Knowledge Assessment - Tapping Human Expertise by the Query Routine. International Journal of Human-Computer Studies 40 (1), pp.119-151.

Kashihara,A., Hirashima,T., Toyoda,J. (1994). A Cognitive Load Application in Tutoring. User Modeling and User-Adapted Interaction 4 (4), pp.279-303.

Kay,J. (2000). Stereotypes, Student Models and Scrutability. In: Gauthier,G., Frasson,C., VanLehn,K., (Eds.), Lecture Notes in Computer Science 1839, Proceedings of the Fifth International Conference on Intelligent Tutoring Systems, ITS 2000, Montréal, Canada. Springer-Verlag, Berlin, Heidelberg, New York, pp.19-30.

Kearsley,G. (1987). Artificial Intelligence and Instruction - Applications and Methods. Addison-Wesley, Reading, Massachusetts.

Koppen,M. (1993). Extracting Human Expertise for Constructing Knowledge Spaces: An Algorithm. Journal of Mathematical Psychology 37, pp.1-20.

Langley,P., Ohlsson,S. (1984). Automated Cognitive Modelling. Proceedings of the National Conference on Artificial Intelligence, Austin, Texas.

Lassez,C. (1987). Constraint Logic Programming. BYTE August 1987, pp.171-176.

Lee,F.L. (1996). Electronic Homework: An Intelligent Tutoring System in Mathematics (PhD Thesis), The Chinese University of Hong Kong.

Levesque,H.J. (1986). Knowledge Representation and Reasoning. Annual Review of Computer Science 1, pp. 255-287.

Lightfoot,J.M. (1999). Expert knowledge acquisition and the unwilling expert: a knowledge engineering perspective. Expert Systems 16 (3), pp.141-147.

Linn, R. L., Baker, E. L., & Dunbar, S. B. (1991). Complex, performance-based assessment: Expectations and validation criteria. Educational Researcher, 20(8), pp.15-21.

Llewellyn,S., Greer,A. (1996). Mathematics - The Basic Skills, Fifth Edition. Stanley Thornes, Chelternam.

Lord,F.M. (1980). Applications of Item Response Theory to Practical Testing Problems. Lawrence Erlbaum Associates, Publishers, New Jersey.

Mandl,H., Lesgold,A. (1988). Learning Issues for Intelligent Tutoring Systems. Springer-Verlag, New York.

Mao,Y., Lin,J. (1992). Intelligent Tutoring System for Symbolic Calculation. In: Frasson,C., Gauthier,G., McCalla,G.I., (Eds.), Intelligent Tutoring Systems. Second International Conference, ITS'92, Montréal, Canada, June 1992 proceedings. Lecture Notes in Computer Science 608. Springer-Verlag, Berlin, Heidelberg, New York, pp.132-139.

Marriott,K., Stuckey,P. (1998). Programming with Constraints: An Introduction. MIT Press, Cambridge, MA.

Marshall,S.P. (1981). Sequential Item Selection: Optimal and Heuristic Policies. Journal of Mathematical Psychology 23, pp.134-152.

Marshall,S.P. (1990). Generating Good Items for Diagnostic Tests. In: Frederiksen,N., Glaser,R., Lesgold,A., Shafto,M.G., (Eds.), Diagnostic Monitoring of Skill and Knowledge Acquisition. Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, Hove, London, pp. 407-433.

Matsubara,Y., Nagamachi,M. (1996). Motivation System and Human Model for Intelligent Tutoring. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Lecture Notes in Computer Science 1086. Proceedings of the Third International Conference on Intelligent Tutoring Systems, ITS'96, Montreal, Canada, June 12-14. Springer, Berlin, Heidelberg, New York, pp. 139-147.

McArthur,D. (1994). Some Possible Futures For Artificial Intelligence In Mathematics Education. http://www.rand.org/hot/mcarthur/Papers/future.html

McCalla,G., Greer,J., Barrie,B., Pospisil,P. (1992). Granularity Hierarchies. Computers & Mathematics with Applications 23 (2-5), pp.363-375.

McCalla,G.I. (1992). The Central Importance of Student Modelling to Intelligent Tutoring. In: Costa,E., (Ed.), New Directions for Intelligent Tutoring Systems. Springer-Verlag, pp. 107-131.

McCalla,G.I., Greer,J.E. (1994). Granularity-Based Reasoning and Belief Revision in Student Models. In: Greer,J.E., McCalla,G.I., (Eds.), Student Modelling: The Key to Individualized Knowledge-Based Instruction. Springer-Verlag, Berlin Heidelberg, pp. 39-62.

McGraw,K.L., Harbison-Briggs,K. (1989). Knowledge Acquisition, Principles and Guidelines. Prentice-Hall, Englewood Cliffs, N.J., London.

Mertz, J.S. (1997). Using a simulated student for instructional design, International Journal of Artificial Intelligence in Education, 8, pp.116-141.

Microsoft (2000). Adaptive Testing. http://www.windowsgalore.com/cert/adaptive_testing/index.htm

Miller, G.A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review, 63,* 81-97.

Mitrovic,A. (1998). Experiences in Implementing Constraint-Based Modeling in SQL-TUTOR. In: Goettl,B.P., Halff,H.M., Redfield,C.L., Shute,V.J., (Eds.), Intelligent Tutoring Systems. 4th International Conference, ITS'98, San Antonio, Texas, USA, August 1998 Proceedings. Lecture Notes in Computer Science 1452. Springer, Berlin, Heidelberg.

Mizoguchi,R. (2000). Workshop on Ontological Engineering and its Implications to AIED Research, held as part of the Fifth International Conference on Intelligent Tutoring Systems, ITS'2000, June 19, Montréal, Canada.

Mizoguchi,R., Bourdeau,J. (2000). Using Ontological Engineering to Overcome Common AI-ED Problems. International Journal of Artificial Intelligence in Education 11, pp.107-121.

Morales,R., Pain,H., Conlon,T. (1999). Workshop on Open, Interactive and Other Overt Approaches to Learner Modelling, held as part of 9th International Conference on Artificial Intelligence in Education. Open Learning Environments: New Computational Technologies to Support Learning, Exploration and Collaboration, AI-ED 99, July 19-23, LeMans, France.

Murray, W.R. (1998). A Practical approach to Bayesian Student Modelling. In: Goettl,B.P., Halff,H.M., Redfield,C.L. & Shute,V., (Eds.), Intelligent Tutoring Systems: Proceedings of 4th International Conference, ITS'98, San Antonio, Texas, USA, August 1998, No.1452 Lectures Notes in Computer Science. Springer, Berlin, Heidelberg, pp.425-433.

Murray,T. (1999). Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art. International Journal of Artificial Intelligence in Education 10, pp.98-129.

Murray,T. (2000). Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems. Preface to the IJAIED Special Issues on Authoring Systems for Intelligent Tutoring Systems.

Neapolitan, R. (1990). Probabilistic Reasoning in Expert Systems. Theory and Algorithms. Wiley InterScience.

Niedermayer, D. (1998). An Introduction to Bayesian Networks and their Contemporary Applications, http://www.gpfn.sk.ca/~daryle/papers/bayesian_networks/bayes.html#theorem

Nwana,H.S. (1993). The Anatomy of FITS: A Mathematic Tutor. In: Nwana,H., (Ed.), Mathematical Intelligent Learning Environments. Intellect Book, pp. 403-408.

Ohlsson,S. (1987). Some Principles of Intelligent Tutoring. In: Lawler, Yazdani, (Eds.), Artificial Intelligence and Education, Volume 1. Ablex, Norwood, NJ, pp. 203-238.

Ohlsson,S. (1994). Constraint-Based Student Modeling. In: Greer,J.E., McCalla,G.I., (Eds.), Student Modelling: The Key to Individualized Knowledge-Based Instruction. NATO ASI Series. Series F: Computer and Systems Sciences, Vol.125. Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, Barcelona, Budapest, pp. 167-189.

Okamoto,T. (1994). The Current Situations and Future-Directions of Intelligent CAI Research/Development. IEICE Transactions on Information and Systems E77D (1), pp. 9-18.

Oseas-Europe (2000). http://www.bibl.u-szeged.hu/oseas/papcomp.html

Paiva,A. (1995). Dynamic User and Learner Modelling (PhD Thesis), University of Lancaster, UK.

Park,O.-C. (1996). Adaptive Instructional Systems. In: Jonassen,D.H., (Ed.), Handbook of Research for Educational Communications and Technology. A Project of the Association for Educational Communications and Technology. Simon & Schuster Macmillan, New York, pp. 634-664.

Parvate,V., Rajan,P., Anjaneyulu,K.S.R. (1998). Mathemagic: An Adaptive Remediation System for Mathematics. Journal of Computers in Mathematics and Science Teaching 17 (2/3), pp.265-284.

Payne,S.J., Squibb,H.R. (1990). Algebra Mal-Rules and Cognitive Accounts of Error. Cognitive Science 14 (3), pp.445-481.

Pearl, J. (1988). Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann.

Polson,M.C., Richardson,J.J. (1988). Foundations of Intelligent Tutoring Systems. Lawrence Erlbaum Associates, Hillsdale, New Jersey, Hove and London, pp.191-207.

Pountain,D. (1995). Constraint Logic Programming. BYTE , February issue.

Putnam,R.J. (1987). Structuring and Adjusting Content for Student: A Study of Live and Simulated Tutoring of Addition. American Educational Research Journal 24 (1).

Renkl,A. (1997). Learning from Worked-Out Examples: A Study on Individual Differences. Cognitive Science 21 (1), pp.1-29.

Ritter,S., Brusilovsky,P., Medvedeva,O. (1998). Creating more versatile intelligent learning environments with a component-based architecture. Lecture Notes in Computer Science 1452, pp.554-563.

Ritter,S., Koedinger,K.R. (1996). An Architecture for Plug-in Tutor Agents. Journal of Artificial Intelligence in Education 7 (3/4), pp.315-347.

Ríos,A., Millan,E., Trella,M., Perez-de-la-Cruz, Conejo,R. (1999). Internet Based Evaluation System. In: Lajoie,S.P., Vivet,M., (Eds.), Artificial Intelligence in Education. Open Learning Environments: New Computational Technologies to Support Learning, Exploration and Collaboration. Volume 50 in Frontiers in Artificial Intelligence. IOS Press, Amsterdam, pp. 387-394.

Rocklin,T., O'Donnell,A.M. (1987). Self-Adapted Testing: A Performance-Improving Variant of Computerized Adaptive Testing. Journal of Educational Psychology 79 (3), pp.315-319.

Rocklin,T.R. (1994). Self-Adapted Testing. Applied Measurement in Education 7 (1), pp.3-14.

Ross,B.H., Kennedy,P.T. (1990). Generalizing from the Use of Earlier Examples in Problem Solving. Journal of Experimental Psychology-Learning Memory and Cognition 16 (1), pp.42-55.

Rudner,L.M. (1998). An On-line, Interactive, Computer Adaptive Testing Mini-Tutorial, 11/98, http://ericae.net/scripts/cat/catdemo.htm

Russell, S. and Norvig, P. (1995) Artificial Intelligence: A Modern Approach, Prentice Hall.

Seidel,R.J., Park,O.-C. (1994). A Historical Perspective and a Model for Evaluation of Intelligent Tutoring Systems. Journal of Educational Computing Research 10 (2), pp.103-128.

Self,J.A. (1974). Student Models in Computer-Aided Instruction. International Journal Man-Machine Studies 6, pp.261-276.

Self,J.A. (1979). Student Models and Artificial Intelligence. Computers & Education 3, pp.309-312.

Self,J. (1988a). Artificial Intelligence and Human Learning - Intelligent Computer-aided Instruction. Chapman and Hall, London, New York.

Self,J. (1988b). Student Models: What use are they? In: Ercoli,P., Lewis,R., (Eds.), Artificial Intelligence Tools in Education. Elsevier Science Publishers B.V., North Holland, pp. 73-86.

Self,J. (1990). Bypassing the Intractable Problem of Student Modeling. In: Frasson,C., Gauthier,G., (Eds.), Intelligent Tutoring Systems: At The Crossroads of Artificial Intelligence and Education. Ablex Publishing Corporation, Norwood, NJ, pp. 107-123.

Self,J. (1994). Formal Approaches to Student Modelling. In: Greer,J.E., McCalla,G.I., (Eds.), Student Modelling: The Key to Individualized Knowledge-Based Instruction. Springer-Verlag, Berlin Heidelberg, pp. 295-352.

Self,J. (1995). Computational Mathetics: Towards a Science of Learning Systems Design. www.cbl.leeds.ac.uk/~jas/cm.htm

Self,J. (1999a). Open Sesame?: Fifteen Variations on the Theme of Openness in Learning Environments. International Journal of Artificial Intelligence in Education 10, pp.1020-1029.

Self,J. (1999b). The Defining Characteristics of Intelligent Tutoring Systems Research: ITSs Care, Precisely. International Journal of Artificial Intelligence in Education 10.

Shute, V.J. (1995). SMART: Student Modeling approach for responsive tutoring. User modeling and user-adapted instruction, 5, pp.1-44.

Shute,V.J., Psotka,J. (1996). Intelligent Tutoring Systems: Past, Present, and Future. In: Jonassen,D.H., (Ed.), Handbook of Research for Educational Communications and Technology. A Project of the Association for Educational Communications and Technology. Simon & Schuster Macmillan, New York, pp. 570-600.

SICStus (2002). http://www.sics.se/isl/sicstus/docs/3.7.1/html/sicstus_toc.html, http://www.sics.se/isl/sicstus/docs/3.7.1/html/sicstus_33.html#SEC249.

Skinner,B.F. (1954). The Science of Learning and the Art of Teaching. Harvard Educational Review 24, pp. 86-97.

Sleeman,D. (1984). An Attempt to Understand Students' Understanding of Basic Algebra. Cognitive Science 8, pp. 387-412.

Sleeman,D. (1985). Basic Algebra Revisited - A Study with 14-Year-Olds. International Journal of Man-Machine Studies 22 (2), pp.127-149.

Sleeman,D., Kelly,A.E., Martinak,R., Ward,R.D., Moore,J.L. (1989). Studies of Diagnosis and Remediation with High School Algebra Students. Cognitive Science 13, pp.551-568.

Sleeman,D.H., Brown,J.S. (1979). Editorial: Intelligent Tutoring Systems. International Journal of Man-Machine Studies 11, pp.1-3.

Sleeman,D.H., Brown,J.S. (1982). Intelligent Tutoring Systems. Academic Press, London.

Sleeman,D.H., Smith,M.J. (1981). Modelling Student's Problem Solving. Artificial Intelligence 16, pp.171-188.

Soloway,E.M., Johnson,W.L. (1984). Remembrance of Blunders Past: A Retrospective on the Development of PROUST. Proceedings of the Sixth Cognitive Science Society Conference, Boulder, CO 57.

Stern,M., Beck,J., Woolf,B.P. (1996). Adaptation of Problem Presentation and Feedback in an Intelligent Mathematics Tutor. In: Frasson,C., Gauthier,G., Lesgold,A., (Eds.), Lecture Notes in Computer Science 1086. Intelligent Tutoring Systems - Third International Conference, ITS '96, Montreal, Canada, June 1996 proceedings. Springer, Berlin, Heidelberg, pp. 605-613.

Suppes,P. (1966). The Uses of Computers in Education. Scientific American 215 (2), pp.206-220.

Sweller,J. (1988). Cognitive Load During Problem Solving: Effects on Learning. Cognitive Science 12, pp. 257-285.

Syang,A., Dale,N.B. (1993). Computerized adaptive testing in computer science: assessing student programming abilities. Proceedings of the twenty-fourth SIGCSE technical symposium on Computer Science Education.February 18-19 1993.Indianapolis USA, pp.53-56.

Taylor,J.A. (1998). Self Test: a flexible self assessment package for distance and other learners. Computers & Education 31 (3), pp.319-328.

Thissen,D., Mislevy,R.J. (1990). Testing Algorithms. In: Wainer, H. Computerized Adaptive Testing: A Primer. Lawrence Erlbaum Associates, Publishers, New Jersey, pp. 103-135.

TOEFL (2000). http://www.toefl.org

Tsang,E. (1993). Foundations of Constraint Satisfaction. Academic Press, London, San Diego, New York, Boston, Sydney, Tokyo, Toronto.

Uhr,L. (1969). Teaching Machine Programs that Generate Problems as a Function of Interaction with Students. Proceedings of the 24th National Conference, pp.125-134.

Ur, S. and VanLehn, K. (1995). STEPS: A simulated, tutorable physics student. Journal of Artificial Intelligence and Education, 6(4), pp.405-437.

Urban-Lurain,M. (1996). Intelligent Tutoring Systems: An Historic Review in the Context of the Development of Artificial Intelligence and Educational Psychology. http://web.cps.msu.edu/~urban/ITS.html

vanderLinden,W.J. (1998). Bayesian Item Selection Criteria for Adaptive Testing. Psychometrika 63 (2), pp. 201-216.

VanLehn,K. (1982). Bugs are not enough: Empirical Studies of Bugs, Impasses and Repairs in procedural Skills. Journal of Mathematical Behaviour 3, pp. 3-72.

VanLehn,K (1990). Mind Bugs: The Origins of Procedural Misconceptions. MIT Press., London.

VanLehn,K., Martin,J. (1997). Evaluation of an Assessment System based on Bayesian Student Modeling. International Journal of Artificial Intelligence in Education 8, pp. 179-221.

VanLehn, K. & Niu, Z. (2001). Bayesian student modeling, user interfaces and feedback: A sensitivity analysis. International Journal of Artificial Intelligence in Education, 12.

VanLehn, K., Ohlsson, S., & Nason, R. (1994). Applications of simulated students: An exploration. Journal of Artificial Intelligence and Education, 5(2), pp.135-175.

Villano,M. (1992). Probabilistic Student Models - Bayesian Belief Networks and Knowledge Space Theory. In: Frasson,C., Gauthier,G., McCalla,G.I., (Eds.), Intelligent Tutoring Systems. Second International Conference, ITS'92, Montréal, Canada, June 1992 proceedings. Lecture Notes in Computer Science 608. Springer-Verlag, Berlin, Heidelberg, New York, pp. 491-498.

Virvou,M., Moundridou,M. (2000). Modelling the Instructor in a Web-Based Authoring Tool for Algebra-Related ITSs. In: Gauthier,G., Frasson,C., VanLehn,K., (Eds.), Intelligent Tutoring Systems: Proceedings of 5th International Conference, ITS 2000, Montreal, Canada, No.1839 in Lectures Notes in Computer Science. Springer, Berlin, Heidelberg, New York, pp. 635-644.

Vispoel,W.P., Rocklin,T.R., Wang,T. (1994). Individual Differences and Test Administration Procedures: A Comparison of Fixed-Item, Computerized-Adaptive, and Self-Adapted Testing. Applied Measurement in Education 7 (1), pp.53-79.

Wainer,H. (1990). Computerized Adaptive Testing: A Primer. Lawrence Erlbaum Associates, Publishers, New Jersey.

Wainer,H., Mislevy,R.J. (1990). Item Response Theory, Item Calibration and Proficiency Estimation. In: Wainer,H., (Ed.), Computerized Adaptive Testing: A Primer. Lawrence Erlbaum Associates, Publishers, New Jersey, pp. 65-102.

Weiss,D.J., Kingsbury,G.G. (1984). Application of Computerized Adaptive Testing to Educational Problems. Journal of Educational Measurement 21 (4), pp.361-375.

Welch,R.E., Frick,T.W. (1993). Computerized Adaptive Testing in Instructional Settings. Educational Technology Research & Development 41 (3), pp.47-62.

Wenger,E. (1987). Artificial Intelligence and Tutoring Systems. Computational and Cognitive Approaches to the Communication of Knowledge. Morgan Kaufmann Publishers, California.