Open universiteit

Management, Science & Technology

Software engineering

Towards lightweight student modelling for Functional Programming Tutors

Master thesis
To be defended 7-nov-2017

Student:
Johan Eikelboom
835817329

Course:
IM9906
Master Thesis Software Engineering

Chairman:
prof. dr. Johan Jeuring
Supervisor:
dr. Bastiaan Heeren
2nd Supervisor:
dr. Lloyd Rutledge

Abstract

A student model keeps information on student cognitive states, exercise history and personal preferences in Intelligent Tutoring Systems (ITS). An ITS must select exercises that match student skills, yet are challenging enough to keep attention. When a student gets stuck or makes mistakes, the ITS must decide on an intervention. For these tasks the student model must provide inferencing and query capabilities. The student model must be updated after each step in an exercise.

We have developed a lightweight approach to student modelling. This approach contrasts with approaches that use advanced algorithms with high complexity, yet it can make the required inferences with a limited set of rules. We describe how such a model can be used with domain reasoners from the IDEAS project. We develop rules for the "Ten Steps to complex learning" instructional design methodology. We illustrate how these rules work in a functional programming curriculum.

Contents

Chapter 1

Introduction

The internet has changed the pace of development in many fields. Working with constantly changing technology puts pressure on professionals to keep up to date. Knowledge workers need updates on skills every few years and lifelong learning has become a fact of life (Kruchten, 2015). E-learning is a way to meet the growing demand for training and education. E-learning solutions in the form of Computer Based Training (CBT) or Computer Aided instruction (CAI) have existed for several decades (Woolf, 2010).

An Intelligent Tutoring System (ITS) is an E-learning system in which intelligent technology is used to support working on exercises. The tutor gives hints and feedback on steps during the exercises. Intelligent Tutors have been built for several domains, in mathematics, physics, programming in Lisp (Koedinger et al., 1997; Gertner and VanLehn, 2000; Anderson et al., 1989), etc.

Domain reasoners from the IDEAS[1] project are an intelligent technology that can be used in an ITS. These domain reasoners can evaluate if the student is still on the right track with an exercise, diagnose an error or give hints which step to take if the student is stuck. The Ask-Elle Tutor uses domain reasoners to support students with functional programming exercises in the Haskell programming language (Gerdes et al., 2017).

These domain reasoners have a stateless architecture. However, an ITS must keep state to keep track of a students progress with exercises, and knowledge gained. The student model is the part that keeps information per student. Different students may have different learning behaviour, and some need more exercises than others. The student model makes the ITS adaptive to such needs.

The Ten steps/Four component is an instructional design methodology for learning complex skills (van Merriënboer and Kirschner, 2013). In this research we investigate how a student model can be designed for ITS with domain reasoners from the IDEAS project, following the TenSteps methodology.

[1] `http://ideas.cs.uu.nl`

1.1 Prior work

This research is based upon work from different research areas, and aims at integrating concepts from these areas.

1.1.1 Intelligent Tutor Systems and student models

Intelligent Tutors have existed for several decades (Woolf, 2010). One of the first tutors in the area of programming is the Lisp tutor (Anderson et al., 1989).

ITSs maintain data on student progress in a student model. The general principles in student modelling were described by VanLehn (1988) and Brusilovsky (1992, 1994). Reasoning about student knowledge is difficult as we cannot observe student knowledge directly. We can collect evidence of student knowledge by observing external behavior. This may be difficult to interpret. For example, when answering a multiple choice question the student can make a lucky guess or a slip.

Many systems use statistical methods for inferencing in the student model. Knowledge tracing is a statistical inferencing method based on Bayesian statistics (Corbett and Anderson, 1995). Knowledge spaces theory (KST) was developed to improve assessment methods (Doignon and Falmagne, 2016). KST is based on knowledge states that have prerequisite relationships, for efficient selection of questions in an assessment. A student model can use the same principles for selecting exercises, hints or feedback.

Many researchers use semantic web technology to build the student model as ontology (Ameen et al., 2012). Semantic web technology uses description logic for reasoning with the open world principle (Baader et al., 2007). The open world principle assumes that it must reason with incomplete information. This contrasts with the closed world reasoning were it is assumed that all information is available. Students often acquire much knowledge outside the ITS, e.g. by searching the internet or reading books. When the ITS has incomplete information on student knowledge, an open world reasoner is needed.

Student Modelling is a subtopic in the area of User Modelling. A user model keeps track of the users preferences and cognitive states, that are derived from the interaction history. The user model is used to make software adaptive to user needs by generating help or recommendations, for example in web shops, wordprocessors or music players. User modelling in intelligent user interfaces is an active research area (Brusilovsky and Millán, 2007).

Student modelling is a large research area. Chrysafiadi and Virvou performed a literature review on approaches for student modelling (Chrysafiadi and Virvou, 2013). VanLehn described the behaviour of Tutoring systems in terms of an inner and outer loop. The inner loop uses intelligent technology for conducting an exercise. The outer loop selects exercises and guides the student through te curriculum (VanLehn, 2006). The student model helps to make these loops adaptive to student needs. In a later article loops were described as feedback loops (VanLehn, 2016).

Developing a student model has software engineering dimensions, such as requirements analysis and architecture (Hatzilygeroudis and Prentzas, 2004; Jeremić and Devedžić,

2004). Any student model starts with log data. The PSLC DataShop collects logdata from ITSs in many domains (VanLehn et al., 2007). This requires attention to qualities such as security, privacy and standardisation of dataformats.

1.1.2 Educational theories for ITS design

Theories of learning play an important role in the development of any ITS, and vice versa ITSs have also inspired the forming of learning theories. The Lisp tutor was based on the ACT-R theory of cognition (Anderson, 1996). Problem solving tasks can be carried out by executing a set of rules. The human mind must contain large numbers of rules, some of which are obtained in exercises with an ITS. Novices learn rules and apply them step by step, whereas experts apply rules automatically (VanLehn, 1996). The exercises in an ITS are designed to automate rule application in the mind.

In this thesis we follow the TenSteps methodology for instructional design. This is a methodology for learning complex skills. The methodology is based on learning theories such as ACT-R.

1.1.3 IDEAS domain reasoners in Functional Programming

The IDEAS project has developed domain reasoners that can track the steps in a solution strategy in an exercise (Gerdes et al., 2010; Heeren and Jeuring, 2014). These domain reasoners can be applied in many domains; logic, geometry, microprocessor programming or imperative programming (Keuning et al., 2014). In this paper we concentrate on domain reasoners in the area of functional programming in Haskell. Many universities teach Haskell to students to become acquainted with functional programming concepts (Hutton, 2016).

The Haskell Expression Evaluator (HEE) is a tutor with which the student can stepwise evaluate Haskell expressions (Olmer et al., 2014). Understanding expression evaluation is essential for the understanding of functional programming. Students with a background in imperative program have to make a mind shift when starting with Haskell.

The Ask-Elle tutor can conduct exercises where the student has to create a small function in Haskell (Jeuring et al., 2014). Ask-Elle can conduct class 3 exercises (Le et al., 2013), where there are multiple solutions strategies (Gerdes et al., 2017).

1.2 Our contribution: lightweight student modelling.

1.2.1 Problem statement and research question

The above mentioned IDEAS tutors are stateless. The tutor does not keep track of which exercises the student has completed. The outer loop of Ask-Elle is a menu of exercises.

As a student progresses through the curriculum we must recommend exercises that are not too difficult and not too easy. The outer loop of an ITS recommends exercises and should adapt to student needs and preferences. A fast student may skip exercises

that are too easy, the system must not insist on completing easier exercises even when they are prerequisite. A slow student may need more exercises.

Information on student progress is kept in a student model. The student model can be queried during exercises, to choose between interventions. Typical interventions are: reminding, persuasion, teaching and remediation (VanLehn, 2016).

Exercises and model solutions must be linked to domain concepts and learning objectives. The model must make it easy for the author to specify exercises.

The student model must be updated after each step. Some inferencing mechanism is needed to infer the students cognitive state from the task step history. Machine learning and Bayesian models are often used as intelligent technology for knowledge inferencing. These technologies require a lot of data and effort to set up and parameterise. We are looking for a lightweight solution for knowledge representation and knowledge inferencing. This must be easy to implement in research and development situations where resources are limited.

The central question in this research is how we can build a lightweight student model that supports the TenSteps methodology and meets the requirements mentioned.

1.2.2 Our solution

An overlay model is a model that overlays a domain model, by assigning a value to each domain concept. Overlay models are often used as the first step for introduction of a student model. Most ITSs use an overlay model in their student model, often combined with other approaches (Chrysafiadi and Virvou, 2013).

We use lattice theory to develop an overlay model and a rule system for knowledge tracing. The ruleset follows the TenSteps methodology, but is not specific for any domain. The author only has to specify properties for exercises.

We show how the problems mentioned above are solved in some scenarios from a functional programming curriculum. We found that the scenarios specified could be implemented with a very limited ruleset. With this approach a student model can be introduced in a new ITS with limited effort.

1.2.3 Thesis organisation

This thesis is organised as follows. Chapter 2 discusses the components that provide the foundation for our work. We combine concepts from the TenSteps methodology, mathematics (lattice theory), domain reasoners and ITS and functional programming. This chapter provides a brief introduction to these concepts. Our method for student modelling and inferencing is explained in chapter 3. In chapter 4 we demonstrate how this works for some scenarios. In chapter 5 we analyse the threats to validity, the conclusions and future work.

Chapter 2

IDEAS Domain Reasoners in an ITS based on TenSteps

In this chapter will look at the TenSteps (van Merriënboer and Kirschner (2013)) methodology as design principle for an ITS.

2.1 Domain reasoners in a functional programming curriculum

2.1.1 Domain reasoners

A domain reasoner guides a student through exercises. Many problems can be solved step by step by substitutions. This principle applies in many domains, such as adding fractions, solving linear equations and programming languages.

Functional programming is also based on this principle. One of the first exercises in Huttons's textbook is the evaluation of a simple function double (Hutton, 2016).

$$double\ x = x + x \tag{2.1}$$

We define a function *maxi* to determine the maximum of two numbers.

$$maxi\ x\ y = if\ x <= y\ then\ y\ else\ x \tag{2.2}$$

The *maxi* function uses a conditional expression and a comparison.

```
double  3
  => expr.double
3  +  3
  => expr.plus
6
```

```
maxi  4   5
   => expr.maxi
if ( 4  <=  5 ) then  5  else  4
   => expr.leq
if  True  then  5  else  4
   => expr.if
 5
```

Listing 1: Derivations of double and maxi

Listing 1 shows how a domain reasoner applies rules to make stepwise derivations for these expressions. Two rules are applied for double: expr.double and expr.plus. For maxi there are three rules: expr.maxi, expr.leq and expr.if. A strategy describes how rules can be applied. For exercises with maxi and double the rules may be applied anywhere possible in the strategy .

```
maxi (double  3) 7
   => expr.double
maxi (3 + 3) 7
   => expr.plus
maxi 6 7
   => expr.maxi
if (6 <= 7) then 7 else 6
   => expr.leq
if True then 7  else  6
   => expr.if
 7
```

```
maxi (double  3)  7
   => expr.double
maxi (3 + 3) 7
   => expr.maxi
if ((3 + 3) <= 7) then 7 else (3 + 3)
   => expr.plus
if (6 <= 7) then 7 else (3 + 3)
   => expr.leq
if  True then 7 else (3 + 3)
   => expr.if
 7
```

Listing 2: Derivations of combined

Expressions may be nested, for example, *double* (*double 3*) or *maxi* (*double 3*) *7*. Listing 2 shows alternative derivations for *maxi* (*double 3*) *7*.

A domain reasoner can use the strategy and expression to offer services such as:

- give a diagnosis when a student enters an answer for a step.
- give a hint about a possible next step in a derivation.
- print a solution.

2.1.2 A functional programming exercise

The next example is a programming assignment in an introductory course on functional programming. In a functional programming course, students learn about functional datastructures such as lists or trees, and programs that process these structures.

The mylength exercise

As an example of a functional programming exercise, an instructor might ask the students to write a function *mylength* to compute the length of a list. The student may find the following solution.

$$
\begin{aligned}
&\mathit{mylength} :: [a] \to \mathit{Int} \\
&\mathit{mylength}\ [\,] = 0 \\
&\mathit{mylength}\ (x : xs) = 1 + \mathit{mylength}\ xs
\end{aligned}
$$

Listing 3: The standard recursive solution

Listing 3 is a typical solution from a textbook. We will assume that the student found the solution himself. In that case there is evidence that the student understands how to define a function type, how to use recursion and pattern matching on lists. These are three items from the domain model. In the student model we will add an indication or increment a counter to mark the understanding of these items.

Students must learn to solve problems in case an error is made. We ask the learner to reason how the expression $\mathit{mylength}[1, 2, 3]$ gets evaluated. Listing 4 shows how the learner evaluates this expression, similar to what he learned in exercises maxi and double.

$$
\begin{aligned}
&\mathit{mylength}\ [1,2,3] = \\
&\mathit{mylength}\ (1 : [2,3]) = \\
&1 + \mathit{mylength}\ (2 : [3]) = \\
&1 + (1 + \mathit{mylength}\ (3 : [\,])) = \\
&1 + (1 + (1 + (\mathit{mylength}\ [\,]))) = \\
&1 + (1 + (1 + 0)) = \\
&1 + (1 + 1) = \\
&1 + 2 = \\
&3
\end{aligned}
$$

Listing 4: Expression evaluation

There is a domain reasoner available that can assist the student with such exercises (Olmer et al., 2014). Programming assignments usually do not have a unique solution. There are many variations that are acceptable.

```
mylength :: [a] → Int
mylength = length
  --
mylength :: [a] → Int
mylength xs = if null xs then 0 else 1 + mylength (tail xs)
  --
mylength :: [a] → Int
mylength = mylength' 0 where
  mylength' n [] = n
  mylength' n (x : xs) = (mylength' $! (n + 1)) xs
  --
mylength :: [a] → Int
mylength = foldl (λx _ → x + 1) 0
```

Listing 5: Some other solutions

All functions in listing 5 are working solutions for the mylength exercise. The first solution proves that the student can use function *length* from the prelude, but nothing else. The second provides evidence for recursion, conditional expressions with 'if' and prelude function *null*. The second solution is tail-recursive, but it is not clear that the student understands that concept. The third solution provides evidence that the student masters pattern matching on lists, where syntax, strict evaluation and recursion. The last solution provides evidence that the student knows of about *foldl* from the prelude and can apply the concept of higher order functions.

The Ask-Elle domain reasoner (Gerdes et al., 2017) supports exercises similar to the mylength exercise. The Ask-Elle domain reasoner can recognize alternative solutions. The domain reasoner can give hints if the student cannot complete the exercise, or even demonstrate a solution strategy. If a student finds a wrong solution the domain reasoner will give a counter-example. For example, if the student sets the length of an empty list to one, then a counter-example is that $mylength[1, 2, 3]$ evaluates to 4.

This example illustrates some principles.

- The student can choose from multiple solution strategies.
- Which solution to expect depends on the learning objectives and where we are in the curriculum.
- Successful completion of an exercise without support is evidence that the student can apply certain concepts. The solution must be analyzed to determine which concepts have been used.
- Domain reasoners must provide procedural support (scaffolding) on demand.

2.1.3 Classes of domain reasoners

We have seen two examples of exercises conducted by domain reasoners: an expression evaluation exercise and programming exercise. These exercises are different. For expression evaluation there is a well defined solution strategy, with well defined rules. For a programming exercise there is no general strategy that always leads to a solution. For these exercises there are expert solutions. A domain reasoner such as Ask-Elle dynamically derives a strategy to lead the student to a solution. Exercises can be classified in five classes from well defined to ill-defined (Le et al., 2013):

1. problems for which there is a single good answer.
2. problems for which a one solution strategy is available, which may be implemented in multiple variants.
3. problems for which some known solution strategies are available.
4. problems for which a great variety of solution strategies are available that can be assessed automatically.
5. problems for which the correctness of a solution cannot be determined automatically.

Many domain reasoners are applied to problems of class 2. For a class 2 problem a domain reasoner can perform accurate model tracing, and verify that the student takes the correct path. The path that the student took can be used to update the student model.

Programming exercises that do not have a single solution strategy are class 3. When the student gets stuck, the domain reasoner can give a hint and lead the student to one of the expert solutions. However, the student may also come up with a perfectly valid solution that is not recognised by the domain reasoner. In contrast to class 2 problems we must not rely on recognising the students solution path to update the student model.

To update the student model we need to know properties of the solution. We can conclude that the student needed help if a hint or model solution was given. Ask-Elle determines one important property: the solution is correct or incorrect. The domain reasoner could determine more properties when parsing the solution: language concepts, syntax and library routines used. Such properties may be used to update the student model.

2.2 The Ten Steps/Four component instructional design methodology

Modern professionals such as surgeons, airline pilots or information system designers often have to carry out complex tasks. An airline pilot has to understand aircraft control and dynamics, interpret meteorologic information, navigate and instruct the crew and

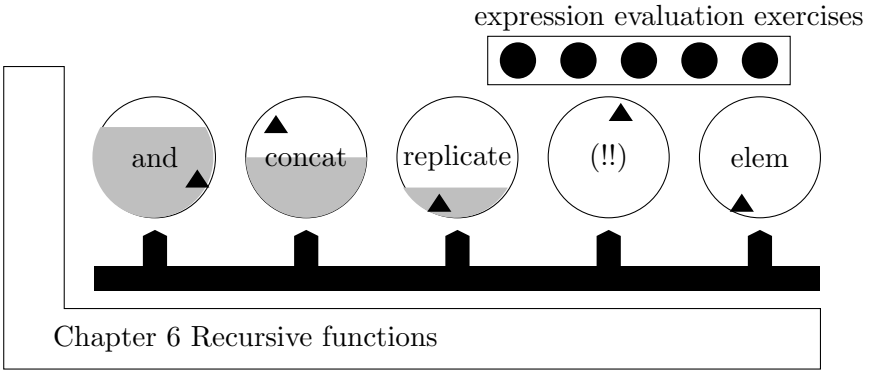

Figure 2.1: Task class Recursive functions

passengers in safety procedures. Similarly surgeons have to understand microbiology, medicines, anatomy, operating procedures, instruments, and how to communicate with patients. Multiple competenties are needed to carry out such complex tasks.

The TenSteps methodology methodology aims at designing effective instruction for such complex tasks. For training complex skills a holistic design approach is used. Other instruction models often compartmentalize a domain in separate parts, cognitive affective or psychomotoric skills. Fragmentation is the process of breaking a domain in separate pieces that can be covered separately.

If a student has to learn three concepts c1 c2 and c3 then we must choose how to sequence exercises:

per topic c1, c1, c1, c2, c2, c2, c2, c3, c3.

mixed c1, c3, c2, c3, c2, c1, c3, c2, c1, c2.

The first approach takes less time to complete the exercises, which is efficient in the short term. It turns out that the second approach has a better transfer of learning. This is called the transfer paradox.

Avoiding compartmentalization, fragmentation and the transfer paradox is an important aspect of the TenSteps methodology. The TenSteps methodology methodology favours inductive learning where knowledge is gained from concrete experiences. In deductive learning knowledge is gained by reasoning, e.g. from general cases to specific cases.

2.2.1 The four components of the training blueprint

The four components in the TenSteps methodology are depicted in figure 2.1. Figure 2.1 is a possible task class modelled after chapter 6 “Recusive functions” of “Programming in Haskell” (Hutton, 2016). The four components are the following:

- Learning tasks, grouped in task classes.
  Each large circle in figure 2.1 represents a learning task.

- Supportive information.
  The big L-shaped form represents supportive information. In this case the chapter from the textbook is used as supportive information. The supportive material is made available in the beginning, probably supported with a lecture or a video. During the task class this is available at any moment.

- Procedural information on routine aspects of the tasks.
  The novice learner must be shown how to do it. A teacher may demonstrate the task execution, or give hints. This, however, fades away during task class execution, as depicted by the grey areas in each circle.

- Part-task practice to automate task aspects that are repeated often.
  The small black circles depict part-task practice exercises for small aspects that must be carried out automatically.

Learning tasks and task classes

The major building blocks in the Ten Steps method are learning tasks, organized in task classes. Learning tasks must be authentic real-life whole-task learning experiences that integrate multiple skills, knowledge and attitudes.

The small triangles at different angles represent variability of the tasks. Variability is important for the development of schema-based processes in the mind. Within a task class, the learning tasks must vary along the relevant dimensions. For example, in the Lisp tutor it was discovered that students who had learned to define functions with one parameter started to make errors when a function with two parameters was encountered (Corbett and Anderson, 1995). In that case the number of parameters is one dimension that must be varied.

Recursion comes in multiple forms. Examples are recursive algorithms, mutual recursion, multiple recursion, tail recursion and recursion on lists. The example of figure 2.1 contains only exercises with recursion on lists. The learner may get the impression that recursion is connected to lists.

The tasks are grouped in task classes of equal difficulty or complexity. The task class is the unit of sequencing in Ten Steps. Task classes are usually of increasing complexity. There is a prerequisite structure between task classes. This does not necessarily imply a linear structure. It is possible to allow the learner to choose between options as long as prerequisites are met.

Supportive information

The learner receives supportive information when beginning a new task class. This can be a book, a presentation or even the Haskell language report. The supportive information is presented in the beginning of the task class, and available during task execution. The L-shaped form in 2.1 represents supportive information. This will explain the problems in the domain, solution strategies and mental models of how the domain is organised.

Procedural information and procedural support

Procedural information supports the development of rule-based processes in the mind. The tutor provides procedural information just in time as much as is needed. In the beginning a demonstration may be given of the task, and later hints and feedback. As the learner proceeds through the task class he will get more routine. The procedural support then fades away.

The right amount of repetition is an important factor for rule-based processes. When the learner can execute tasks without support he may proceed to a further task class. It is not necessary that everyone completes all the tasks within the task class. An ITS needs a student model to provide this adaptivity.

Part-task practice

Learning tasks, as described above, are to practice real-life whole-tasks. Sometimes more practice is needed on a specific aspect of a complex skill. In the domain of functional programming we may think of code reading and priority rules in expressions. Applying parentheses rules and recognizing wrong or unnecessary use of parentheses is a valuable skill. Part-task practice exercises offer an opportunity to drill these skills to be executed automatically. In figure 2.1 the part task practice is denoted by the small black circles in the right upper corner.

2.2.2 Mapping the TenSteps methodology to ITS design with domain reasoners.

The TenSteps methodology and ITS world share similar concepts and methodologies, but use different terminology. An ITS is one specific area where the TenSteps methodology can be applied.

It is not required to execute all the steps. The four components are designed in ten steps. Some artefacts are developed to support this design. These artefacts can also be used in the design of domain reasoners and student model.

A domain model is made, that contains important concepts and causal relations and structure of the knowledge domain. This domain model can be used for both the student model and the design of supportive information. It is useful if a Systematic Approach to Problem Solving (SAP) is available or can be constructed. Such a SAP can be presented in supportive information and used to design strategies for domain reasoners. Goals and prerequisites must be formulated for task classes and learning tasks, and assessment instruments developed.

In general ITS design and the TenSteps methodology use similar principles, but sometimes different terminology. In the table 2.1 below we show how we apply the TenSteps methodology to ITS design.

Learning tasks

Step	Ten Steps concept	ITS IDEAS concept
1	Learning tasks	Elements of the outer loop, examples in IDEAS domain reasoner exercises
2	Assessment	Diagnose service during exercise execution
3	Sequence learning tasks	Outer loop
4	Supportive information	Text books, references to documentation
5	Cognitive Strategies	Rules and strategies in domain reasoners
5	SAP	Design artefact for strategy and rule design
6	Mental models	Used to structure our domain model: concepts, causal relations, structural relations
7	Procedural support	Domain reasoner can generate a hint, next step, or a worked out solution
8	Cognitive rules	Rules in domain reasoners, if then else rule, buggy rules
9	Prerequisite knowledge	Must be checked automatically
10	Part-Task practice	Small tasks to address an issue e.g. a missing prerequisite or misconception

Table 2.1: Mapping of Ten Steps concepts

2.3 PoSets and Lattices

In the next section we will show how lattices can be used in a student model. This section presents a small review of lattice and poset terminology and properties.

2.3.1 PoSets

A partial ordering is a binary relation $x \leqslant y$ in a set (Saunders MacLane, 1988). A partial ordering has the following properties.

- Transitive, if $a \leqslant b$ and $b \leqslant c$ then $a \leqslant c$.
- Reflexive, the relation $\leqslant$ hold for any element with itself: $a \leqslant a$.
- Anti-symmetric if $a \leqslant b$ and $b \leqslant a$ then $a = b$.

A set with a partial ordering is called a poset. If in a set this $\leqslant$ relation can be established for any two elements then we have a linear ordering. In a partial ordering, there may be pairs of elements (x, y) that are not comparable; neither $x \leqslant y$ or $y \leqslant x$. Some posets have an infimum denoted as $\perp$ or bottom, or a supremum denoted as $\top$ or top. In these case the following properties hold in the poset S:

- $\forall x \in S : \perp \leqslant x$
- $\forall x \in S : x \leqslant \top$

2.3.2 Lattices

Two elements x and y may not be comparable, but sometimes we can find elements that are greater or smaller. If we can find an element a such that $a \leqslant x$ and $a \leqslant y$, then a is a lower bound of x and y. The binary operator meet ($\sqcup$) is defined as $a = x \sqcup y$ where a is the largest lower bound, there is no other element a' such that $a \leqslant a'$ and $a' \leqslant x$ and $a' \leqslant y$.

In a similar fashion we can define the operation join ($\sqcap$): $b = x \sqcap y$ if $x \leqslant b$ and $y \leqslant b$ and there is no other value b' with $b' \leqslant b$ and $x \leqslant b'$ and $y \leqslant b'$.

A poset with a meet and join operation is called a lattice. A well known example of a lattice is the set with dividers of 60. Meet is defined as the greatest common divisor e.g.: $12 \sqcup 10 = 2$, $20 \sqcup 15 = 5$. Join is defined as the smallest common multiple e.g. $12 \sqcap 10 = 60$, $4 \sqcap 6 = 12$.

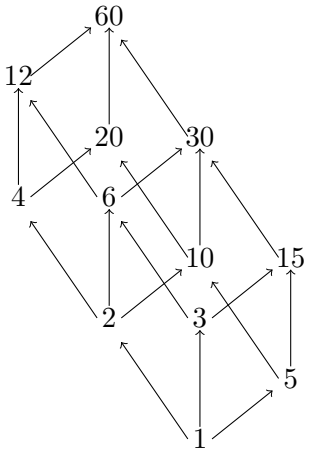

Figure 1: Hasse diagram for dividers of 60

Figure 1 shows a Hasse diagram for this system of dividers of 60. This diagram is based on the concept of immediate successor and predecessor. In this diagram 10 is an immediate predecessor of 30, written as $10 \lll 30$ since there is no element greater than 10 preceding 30. We cannot say $10 \leqslant 15$, in this algebra there is no linear ordering and the numbers are not comparable. In a finite lattice there is always a bottom and a top, in this case 1 and 60.

Boolean values form a lattice. A finite or infinite interval of integer numbers forms a lattice when join is chosen as max and meet as min. Similar any ordered enumeration forms a lattice.

An important property is that lattices can be combined. If A and B are lattices then tuples $(a, b) \in A \times B$ form a lattice.

Equations 2.3 through 2.6 show how meet, join, less or equal and bottom for $A \times B$ can be defined.

$$(a_1, b_1) \sqcap (a_2, b_2) = (a_1 \sqcap a_2, b_1 \sqcap b_2) \tag{2.3}$$

$$(a_1, b_1) \sqcup (a_2, b_2) = (a_1 \sqcup a_2, b_1 \sqcup b_2) \tag{2.4}$$

$$(a_1, b_1) \leqslant (a_2, b_2) = a_1 \leqslant a_2 \wedge b_1 \leqslant b_2 \tag{2.5}$$

$$\bot_{A \times B} = (\bot_A, \bot_B) \tag{2.6}$$

This can be extended to any number of dimensions. If C is a lattice, then in a similar way the set $(a, b, c) \in A \times B \times C$ forms a three dimensional lattice.

Duality

Lattices have a duality property. We can exchange the $\sqcup$ and $\sqcap$, the $\leqslant$ and $\geqslant$ and in finite lattices $\bot$ and $\top$. The result is a new lattice. In a similar fashion alternative definition can be made for equations 2.3 through 2.6. To avoid confusion we will not do this.

Chapter 3

Lightweight reasoning for student models.

If all students would take the same linear learning path, then we could represent student progress by a number. For example, if Mary is working on exercise 6 and Fred on exercise 5, then she is one exercise ahead. Adaptivity to individual needs is a key feature that distinguishes an ITS from other forms of computer aided instruction. Students may take different routes through the material, depending on individual needs and preferences. After exercise 3 the ITS may have given the students the option to choose from exercise 4, 5 and 6. And now we cannot say that Mary is ahead of Fred, but only that both have completed exercise 3.

To measure student progress in an ITS we can use partial orderings. Partial orderings are very common in learning materials. Many authors indicate alternative routes to read chapters based on individual interests.

This chapter describes how lattices can be used to reason about student progress (Section 3.1). The structure of the student model is described in Section 3.2. Section 3.3 describes the rule system in general, and Section 3.2.1 shows how this can be applied in the TenSteps methodology.

3.1 Student model as a lattice.

3.1.1 Progress indicators and progress vectors

An overlay model is the most popular form of student model (Chrysafiadi and Virvou, 2013). It is in the simplest form a list of domain concepts. For each concept a value is assigned as an indication of student progress.

Definition 1. *A progress indicator is a value that ranges over a lattice with a bottom.*

Definition 2. *A progress vector is a set of items. Each item is a (name, value) pair, where the value is a progress indicator.*

In Section 2.3 we have seen that lattices may be combined as tuples to form a new lattice. Thus it follows that a tuple of progress indicators is also a progress vector, and that tuples may be nested to any depth.

In a student model we must keep a large number of progress indicators, and tuples are not practical. Instead of using a number for each dimension we define a set of names $\{name_1, name_2, ..., name_n\}$. A tuple of progress indicators $(pi_1, pi_2, ..., pi_n)$ corresponds to a progress vector $\{(name_1, pi_1), (name_2, pi_2), ..., (name_n, pi_n)\}$.

It is not necessary to store all values, an item is assumed to have a value of bottom if not present. The canonical form is to leave out elements at bottom value.

Progress vectors have a one to one correspondence to tuples. The join, meet, bottom and $\leqslant$ are defined for progress vectors in the same way as for the corresponding tuples. Thus a progress vector is a notational convention for the application of lattices in a student model.

Progress vectors form a lattice with a bottom value, and can be used as progress indicators. This makes the definition recursive. Progress vectors can be nested to any depth.

Definition 3. *Atomic progress indicators are progress indicators that can not be decomposed.*

So our model is made up of atomic progress indicators combined as progress vectors. In Section 3.1.2 we discuss atomic progress indicators.

3.1.2 Atomic progress indicators in the student model.

The dividers of 60 may be an interesting example of the lattice structure, but we will not further use lcm and gcd. In this thesis we work with three types of atomic progress indicators, although many others are possible.

Boolean values (*PiBool*)

Boolean values can be used to indicate that the student masters a concept or an exercise is completed. Boolean values form a lattice, when we take disjunction as join and conjunction as meet operation. Lattices have duality properties, so we could turn this around, but to avoid confusion we will not do so.

Counters (*PiCount*)

Counters are positive integral numbers. Counters form a lattice. We take zero as bottom value, max for join and min for meet. It has a linear ordering, which is a stronger form of ordering than partial ordering.

A special operation is needed to increment or add counters. For this purpose we have added an operator <+>. For counters it adds the counter. For any other progress indicator it is equivalent to join. If we update the student model with this operator, it is still monotonously increasing.

Knowledge level (*PiBloom*)

We need indicators for the levels of learning of domain concepts. Bloom defined a taxonomy with hierarchical levels of learning. This is still used today (Bhalerao et al., 2017). The Bloom taxonomy forms a set with linear ordering:
{Knowledge, Comprehension, Application, Analysis, Synthesis, Evaluation}.
We use this as an enumeration to which we add "Ignorant", and use this as the bottom value.

We must mention that the TenSteps aims at the development of integrated skills. The Bloom metric is somewhat one-sided in that is aims only at the cognitive domain.

Other possible progress indicators

Any datatype that forms a lattice with a bottom value can be used as progress indicator. Booleans, enumerated datatypes or ranges of integers are acceptable as progress indicators. Real values in the interval [0, 1] form a lattice with min and max, and can be used as probabilities.

In Figure 2 we see a set of values that form a partially ordered set with a bottom value (A). This may be a state model $PiState = \{A,\ B,\ C,\ D,\ E,\ F,\ G\}$. It is not a complete lattice. We can see that for example $C \sqcup D = E$ and $E \sqcap G = A$. We can not join any two elements: e.g. $E \sqcup G$ and $B \sqcup F$ are undefined.

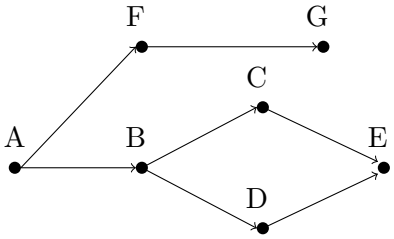

Figure 2: Progress indicators poset PiState

This can be used as progress indicator in the following ways:

- We could add an artificial extra element T following G and E, such that $G \sqcup E = T$.
- We can accept the fact that we have a model constraint and throw an exception when an impossible join (e.g. $F \sqcup C$) is attempted.
- A value from PiState can be represented as a tuple: $\{(v1, v2) \mid v1 \in \{A, F, G\}, v2 \in \{A, B, C, D, E\}\}$ for each path. In this case we also have a model constraint that either $v1$ or $v2$ is bottom. Thus (A, A), (A, D), (G, A) are valid, but (F, B) is an invalid state.
- Another solution is to use boolean indicators to represent the state. The indicators become true as the model reaches greater values. In that case there are many

constraints, for example, if C is true then A and B may be True, but D, F and G must not be true.

Only the first solution conforms completely to our definition of progress indicators. But from a software engineering point of view it makes little difference. In the first case we get a model constraint that values may not assume the value T.

We update the student model with a join operation, so once the student is on path F he cannot reach E anymore. This does not have to be a problem, for example if it is an end state when an exercise is finished.

Some examples with atomic progress indicators

We allready mentioned that an important property of lattices is that they can be combined, to form another lattice. The tuples $(v1, v2, v3) \in PiBool \times PiCounter \times PiBloom$ form a lattice. For example:

- $bottom = (False, 0, Ignorant)$
- $(False, 4, Analysis) \sqcup (True, 5, Comprehension) = (True, 5, Analysis)$
- $(False, 4, Analysis) \sqcap (True, 5, Comprehension) = (False, 4, Comprehension)$
- $(False, 3, Comprehension) \leqslant (True, 5, Analysis) = True$
- $comparable\ \ (False, 4, Analysis)\ \ (True, 5, Comprehension) = False$

3.1.3 Progress vectors for a student model

In step four of the TenSteps methodology a domain model is developed, that describes the knowledge domain. Knowledge domains may contain concepts and terminology, structures and causal relationships. In the domain of functional programming the syntax and prelude functions have a certain structure. Mechanisms such as strict and lazy evaluation are causal relationships.

An overlay model uses a set of elements that cover the knowledge domain. In the simplest form a boolean value is assigned to each element (Brusilovsky, 1992). We use progress indicators in our student model. Our student model is an overlay model in the form a progress vector. The student model may contain elements from the domain model, as well as indicators of exercise progress.

In table 3.1 we see (part of) a progress vector with student progress. We have indicated the type of each atomic indicator.

Key	Progress indicator value	type
DM.Recursion	Knowledge	PiBloom
CAUS.SimpleExpressionEvaluation	Application	PiBloom
SYN.IFexpression	Knowledge	PiBloom
PRE.GTR	Knowledge	PiBloom
PRE.Plus	Comprehension	PiBloom
RL.expr.plus	1	PiCount
CS.double	True	PiBool

Table 3.1: Example of a progress vector for an overlay student model.

In practice this might well contain hundred or more items. The table contains indicators for knowledge levels for domain concepts. This student has read materials on recursion, and he can evaluate simple expression. He applied a rule expr.plus once and he has successfully completed an exercise of type double. The value type should be obvious: True for PiBool, Comprehension is PiBloom and numbers are counters.

Naming conventions

As table 3.1 shows, we have applied a naming convention using prefixes for different kind of indicators.

Prefix	Used for
TC	Task class
LT	Learning task (exercise)
ES	Educational indicators
EX	Exercise type
RL	Rewrite rules
DR	Domain reasoner
PRE	Prelude
DM	Domain concepts
SYN	Syntax

Table 3.2: Prefixes used in naming convention.

Table 3.2 shows the prefixes we use. We will work with deeply nested vectors. We use slashes to indicate the pathname of a nested value or vector in the model, similar as for filenames. The student model has an absolute position of

"$/ES.student$".

An absolute pathname for an exercise in a task class might be:

"$/TC.01.ExpressionEvaluation/LT.01.double$".

We often use the term exercise instead of learning task. In rules we use variables that range over exercise positions, these are written as $\$exercise$.

Many traditional ITSs separate student knowledge, educational progress and misconceptions (Brusilovsky, 1992), but we use just one vector. Any author is free to choose a different structure and naming conventions.

Updating the student model

An item corresponds to a domain concept or progress with exercises and task classes.

The successful completion of an exercise provides evidence of a certain level of mastery of some concepts that form a progress vector. If we denote the student model after exercises n as SM_n and the progress vector as e_n, then we can update the student model:

$$SM_n = SM_{n-1} \sqcup e_n \tag{3.1}$$

Equation 3.1 makes the student model a monotonous function of n:

$$SM_{n-1} \leqslant SM_n \tag{3.2}$$

We have required that progress indicators have a bottom value and this make it easy to start with:

$$SM_0 = \bot \tag{3.3}$$

A top value is not needed, as some progress indicators have a range wider then required. We may not have material in the ITS to reach the level "Evaluation" for every topic. We will be satisfied if the student executes a rule two or three times, however, our counters can easily count up to a million.

3.2 Student model and educational model

A traditional architecture for an ITS consists of a student model and an educational model as separate services. The services might be implemented in separate web or REST services. A separation of concerns is definitely a good idea.

In this thesis, however, we will put everything together in one variable of the type progress vector. We will refer to this as the model. The model has a hierarchical structure as shown in Figure 3.

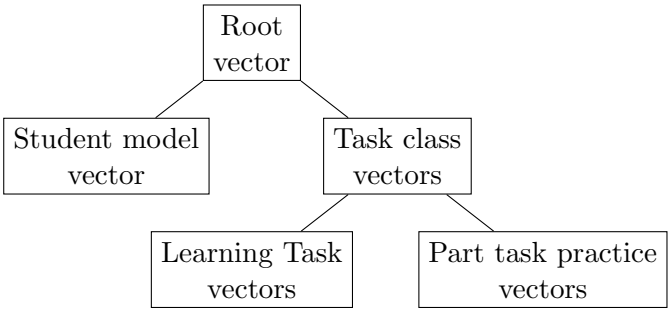

Figure 3: Hierarchical structure of model components

The left branch contains the student model as discussed in Section 3.1.3. The model contains the progress of only one student. The right branch is the educational model, which is decomposed in multiple task classes. Each task class contains multiple learning tasks and part-task practices. This is modelled after the blueprints from the TenSteps methodology.

The whole tree is build up from nested progress vectors. From now on we will often prefix vector by their usage. The root vector is the root of the structure. It contains the vector for the student model, the task class vectors, which contain learning task vectors and part task practice vectors.

The root vector is a progress indicator and can be joined and compared with similar shaped structures. A step vector is such a structure, used to update the model after each task step. A step vector is derived from a request-response cycle with the diagnose service from a domain reasoner.

The step vector has the same structure as shown in Figure 3. A step vector might theoretically contain a student root vector to update the student model directly. In the scenarios we use step vectors with a task class vector and a learning task vector with subvectors. We use rules that evaluate the exercise vector to update the student model vector.

We use an add (<+>) operation to update the model, to be able to count strategy and rule executions. Step vectors may also result from other interactions e.g. multiple choice questions.

Similar like the student model, the whole model is updated with join operations and therefore monotonically increasing. The model mixes progress information with meta-data, such as objectives and prerequisites. It is possible to have variations in metadata per student. For example certain students may benefit from doing more exercises than others.

Another alternative way to look at the model is as an ontology model. The student and the educational components can be seen as individuals in an ontology model. These individuals have relationships and data properties that are here implemented in the structure of the progress vector. This model can be queried and inferences about student progress can be made by a rule system.

3.2.1 Contents of a learning task vector

Name	Type	Description
DR.Rules	Vector	Rewrite rule executions
ES.successCriteria	Vector	Minimum rule executions
ES.objective	Vector	Objectives of task
ES.prerequisite	Vector	Prerequisites of the task
DR.created	PiBool	The exercise is started
ES.finished	PiBool	The exercise is finished
ES.success	PiBool	The exercise is finished successfully
ES.failure	PiBool	The student did not complete the exercise
ES.error	PiBool	The student made errors during the exercise

Table 3.3: Subcontent of a learning task vector.

Table 3.3 details the content of a learning task vector. The first two elements are related. A step vector updates DR.Rules on each rule execution. The ES.successCriteria is set up by the author and defines the minimum number of rule executions. So if $DR.Rules \geqslant ES.successCriteria$ he did execute the rules expected, or otherwise he may have asked hints.

The prerequisites and objectives can be compared or joined with the student model. If $ES.student \geqslant ES.prerequisite$ then the student meets the prerequisites (ES.student denotes the student model vector). If $ES.student \geqslant ES.objective$ then the exercise may be too easy.

The booleans indicate the state of the exercise. The DR. created is set by a step vector when the exercise is started. The other flags are inferred by rules depending on how the exercise was completed. The rules will be discussed later.

3.3 Lightweight inferencing with rules

It is not enough to add up counters of rule executions to update the student knowledge. Some intelligent technology, a rule system in this case, is used to infer updates to the student model.

Rule structure

The rule system that we propose to reason about the student model works similar to other rule systems. A rule has the following form:

$$p_1, .., p_n \rightarrow c_1, .., c_m \tag{3.4}$$

The $p_1, .., p_n$ are a set of predicates and the $c_1, .., c_m$ are conclusions. A predicate has variables that range over the individuals in the ontology. We write commas between the

predicates, when the rule is executed the predicates are evaluated and combined with a logical conjunction. When all predicates evaluate to *True*, the rule fires.

The conclusions are progress vectors in the shape of the model. One rule may have multiple conclusions. The conclusions from all rules that fire are joined into a single conclusion. Often a part of the model is copied somewhere else to form a conclusion. For example, when an exercise is completed successfully we join the exercise objectives to the student model. In our system we have bottom as a zero-element, so if some predicate fails we can say that we conclude bottom.

Some examples of predicates are the following:

- $isTaskClass(?tc)$, $isLearningTask(?lt)$, $isStudent(?st)$
  Individual *tc* is a task class *and st* is the student *and lt* a learningtask. The question marks denote the variables for individuals (after SWRL/OWL). The individuals range over the the whole ontology; this way we make a selection, in fact a cross product of three subsets. These are examples of unary predicates.

- $belongsTo(?lt, ?tc)$
  An example of a binary predicate indicating we only want learningtasks that belong to the task class.

- $isLearningTask(?lt)$, $prerequisites(?lt, ?pre)$, $?st \geqslant ?pre$
  The item *lt* is a learningtask *and lt* has prerequisites *pre and* some item *st* is greater then *pre*. If *st* is a student model, then the student meets the prerequisites of *lt*, as described in section 3.2.1.

- $isLearningTask(?lt)$, $exerciseprogress(?lt, ?ep)$, $succescriteria(?lt, ?sc)$, $?ep \geqslant ?sc$
  The learning task *lt* has made a progress of *ep* and success criteria *sc*. The exercise is completed successfully, as $?ep \geqslant ?sc$.

These predicates may be used in rules, but it is also possible to query the model with predicates, for example to select exercises.

3.3.1 The loop and rule implementation in Haskell

Rule systems as well as databases have advanced mechanisms to work efficiently. Having multiple nested loops ranging over all individuals is expensive. When assertions are added to the ontology, this may enable other rules to fire and require an extra run.

In other rule systems users may be required to prioritise rules as rules may influence each other. OWL/SWRL is designed such that the rules can be run until a fixpoint is reached. This creates restrictions, for example, we cannot update or delete assertions. The rules are monotone in the sense that they only add facts.

In Haskell we do not have an advanced completely generic rule engine. It turns out that we can do most of the reasoning if we iterate over the task classes and learning tasks. Each rule is called with a tuple of four parameters: (root, student, taskclass, learningtask). The conclusions we derive are progress vectors with the same shape as the model. After the rules execute we join all conclusions to the model. Because rules

may affect each other we run this mechanisme a number of times until we reach a fixpoint. All our individuals have the same underlying type of being a (Wrapped) progress vector.

This means:

- Predicates have type:
  $RulePredicate :: (Root, Student, TaskClass, LearningTask) \rightarrow Boolean$
  During the iteration each predicate can be evaluated to a True or False value.
- For conclusions we use the type DerivedFact:
  $DerivedFact :: (Root, Student, TaskClass, LearningTask) \rightarrow Root$
  A DerivedFact can be executed during the iteration to return a conclusion in the form of a progress vector.
- $Root$, $Student$, $TaskClass$ and $LearningTask$ are (wrapped) progress vectors.
- The predicates $isTaskClass(?tc)$, $isLearningTask(?lt)$, $isStudent(?st)$, $belongsTo(?lt, ?tc)$ shown in section 3.3 are now implicit, due to the iteration.
- To create rules we define an operator:
  $(=\!>) :: [RulePredicate] \rightarrow [DerivedFact] \rightarrow Rule$
  These rules can be executed in the iteration, with the four parameters.

Our rule engine may be sensitive to rule execution order. There are some strategies that can be followed to prevent unwanted results.

- Rules can be made once-only, e.g some value is set and a predicate prohibits firing the rule again.
- Rules must specify predicates that are disjunct, so that in each case only one specific action takes place.

If a rule keeps deriving the same fact, it does not have to be a problem as the join operation absorbs the derived fact ($a \sqcup a = a$). The rules execute after the student model is updated with the progress vector derived from the domain reasoner response. For some rules it may be necessary to run multiple times until a fixpoint is reached.

3.4 Rules for exercises

In this section we describe the rules used to implement guidelines from the TenSteps methodology.

3.4.1 Exercise states

We have defined a specific progress indicator for exercise state, to keep track of which exercises have been attempted and the outcome. The exercise state is not kept in a single variable, but inferred from the set of boolean indicators that we showed in table

3.3. This works like the example we showed in Figure 2. An exercise has the following state model:

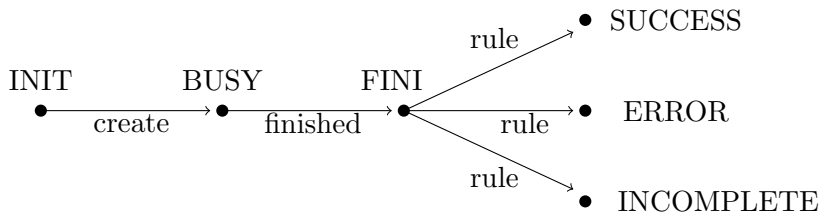

Figure 4: Exercise state

Exercises start in state INIT. When an exercise is started, a create request is send to the domain reasoner. This results in "DR.created" to be set in the exercise. When an exercise is finished the domain reasoner sets a finished flag in the message, resulting in ES.finished being set.

We depend on some component (the user interface perhaps) to send us a finished message when the student stopped with the exercise. The domain reasoner is stateless, and there is no formal exercise termination send to the domain reasoner.

3.4.2 Rules for exercise completion

A rule consists of a set of RulePredicates and a set of DerivedFacts. When all RulePredicates evaluate to true for a given exercise and student knowledge, the DerivedFacts are evaluated to collect the conclusions.

Conclusions have the same form as the model. The first rule below, for example, creates a set indicators in the student part for each indicator in the objectives of the exercise. Effectively it copies the contents of a vector to another place in the model. The engine runs all DerivedFacts and joins the results together.

When an exercise is finished (state FINI) exactly one of four rules may fire:

Success rule :

$/\$exercise/FINI$,
$not\ /\$exercise/DR.rules/ES.error$,
$/\$exercise/DR.Buggy == bottom$,
$/\$exercise/DR.Rules \geqslant /\$exercise/ES.successCriteria$
$\rightarrow$
$/\$exercise/ES.success\ True$ (state becomes SUCCESS),
$\{/ES.student/obj \mid obj \in /\$exercise/ES.objective\}$.

Incomplete rule :

$/\$exercise/FINI$,
$not\ /\$exercise/DR.rules/ES.error$,
$/\$exercise/DR.Buggy == bottom$,
$not\ /\$exercise/DR.Rule \geqslant /\$exercise/ES.successCriteria$

$\rightarrow$
/\$*exercise*/*ES.incomplete True* (state becomes INCOMPLETE).

Error rule :
/\$*exercise*/*FINI*,
/\$*exercise*/*DR.Buggy* == *bottom*,
/\$*exercise*/*DR.rules*/*ES.error*
$\rightarrow$
/\$*exercise*/*ES.error True* (state becomes ERROR).

Buggy rule :
/\$*exercise*/*FINI*,
not /\$*exercise*/*DR.Buggy* == *bottom*
$\rightarrow$
{/*ES.student*/*bug* | *bug* $\in$ /\$*exercise*/*DR.Buggy*},
/\$*exercise*/*ES.error True* (state becomes ERROR).

The success rule must calculate an update to the student model. The rewrite rules that the student executes are counted in a vector with name DR.rules. This is compared with the vector ES.successCriteria to determine successful completion.

The objectives of the exercise are joined to the student model. An exercise double might contain:
"/*TC.01.ExpressionEvaluation*/*LT.01.double*/*ES.objective*/*EX.double True*".
The rule would move this objective to the student model:
"/*ES.student*/*EX.double True*".
After all rules are executed the conclusions are joined to the model.

When the student made a miscalculation or another error, the exercise may be completed. After the exercise is finished, the Error rule fires and puts the exercise in state ERROR. We do not update the student knowledge. The error rule just prevents any of the other rules to run again.

If the student took a shortcut, such as just giving the answer, or just gave up the exercise, it ends in the state INCOMPLETE. Also when a hint is given or other procedural support given the exercise ends in INCOMPLETE. In spite of the fact that the student may probably learn the most from his mistakes, we do not update the model after a procedural support or errors.

The TenSteps methodology prescibes that the student may proceed to a higher task class when he can carry out exercises without requiring procedural support. This is why we only update the student model after a successful exercise completion.

The buggy rule fires in case of a misconception. Domain reasoners may have buggy rewrite rules, that fire when certain errors are recognised. For example, the student should have added a value, but subtracted it. Buggy rewrites are administered in a separate vector DR.Buggy. It is assumed that the exercise will be terminated in this case, so DR.Buggy will only contain one value. We add this to the subvector of the student with bugs. Queries use this information to select an intervention, for example a part-task exercise.

3.4.3 Fast students and Knowledge Space Theory.

The Knowledge Space Theory (KST) is a psychometric theory, that is used for selection of assessment questions (Doignon and Falmagne, 2016), (Reimann et al., 2013). A learning space is defined as a set of knowledge structures that form a partial ordering. A knowledge structure is a subset of learning items that include all prerequisite items. For example, the exercise types from Section 2.1.1 are double, maxi and combined. Exercise double is easier than maxi, which is easier than combined, therefore the recommended order is double, maxi, combined. The sets ks1: {double}, ks2:{double, maxi} and ks3:{double, maxi, combined} are knowledge structures. In a learning space the knowledge stuctures have an inner fringe of structures that have one item less, and an outer fringe with structures that have one item more. For example in our task class, the inner fringe of ks2 is ks1 and outer fringe is ks3.

Suppose a students starts with an advanced exercise of type combined and completes it correctly. It would make little sense to recommend the easier exercises double and maxi.

A rule can be defined to solve this issue. The rule will add the prerequisites of completed exercises to the student model.

Fast student rule :

$/ES.student/ \geqslant /\$exercise/ES.objective$

$\rightarrow$

$\{/ES.student/preq \mid preq \in /\$exercise/ES.prerequisite\}$.

3.4.4 Exercise recommendation query

After all facts have been derived, the model can be queried. For example the user interface can display a list of exercises, that have been completed, with results, or display the student model.

Exercise selection is an important aspect in both ITS and the TenSteps methodology. We may want to select exercises in the outer fringe of the knowledge space. These are exercises for which the student meets the prerequisites, but not yet masters the objectives. The following query can be used to select these exercises.

Exercise recommendation query :

$/ES.student/ \geqslant /\$exercise/ES.prerequisite,$

$not\ /ES.student \geqslant /\$exercise/ES.objective$

Chapter 4

Validation

In section 1.2.1 we raised the question how to build a lightweight student model that can be used with IDEAS domain reasoners. We use the TenSteps methodology as instructional framework. Section 2.2.1 described the four components of the training blueprint. We use the blueprint to structure our educational model (Section 3.1.3).

A student model is derived from a log of the student interactions (VanLehn, 1988). We translate the student interaction in a sequence of step vectors for each task step. We update the model by applying the step vectors in the correct sequence. After each update, the rules are executed to derive new facts, which are joined into the model.

In this chapter we present some scenarios that illustrate how this system can be used. We have created two examples of typical domain reasoners where information from the diagnose service can be used. Section 4.8 presents some arguments why a programming tutor differs from these cases.

4.1 Choice of scenarios

An important characteristic of ITSs is adaptability. A pilot is trained to land an airplane safely in any condition, and it does not matter if it took six or ten lessons. A violin teacher on the other hand may try to get the most out of a talented student.

Curriculum designers must make conscious decisions on adaptability. For example, in a classroom situation it is not practical to have students work on different subjects depending on how fast the are (VanLehn, 2006).

The TenSteps methodology methodology aims at achieving a minimum level of mastery in complex skills. Our student model therefore must adapt to the start situation. Most students complete the curriculum in an average number of lessons. Some students are slower and need some more exercises. One student was sick, missed the previous module and does not meet the prerequisites. Another student took the module but forgot some important details. The next student had to repeat the class, but he already masters this topic. Students may not read carefully and acquire a misconception.

The student model should be able to deal with these problems. The student model makes the ITS a regulating feedback control system similar to a thermostat. The sce-

narios verify that our solution has the potential to meet the requirements. Indeed we abandoned some other attempt because that approach led to more complexity.

The scenario on variations is an exeception, it is not related to the input level. Variability is a principle from the TenSteps methodology.

4.2 Scenario: A normal student

The first exercise is to evaluate the expression *double*(*double*(*double 5*)).

Step	Component	indicator	indicator	value	expression entered
0	LT.01.double	DR.created	True		*double*(*double*(*double5*))
1	LT.01.double	DR.Rules	RL.expr.double	1	*double*(*double*(*5 + 5*))
2	LT.01.double	DR.Rules	RL.expr.plus	1	*double*(*double 10*)
3	LT.01.double	DR.Rules	RL.expr.double	1	*double*(*10 + 10*))
4	LT.01.double	DR.Rules	RL.expr.plus	1	*double 20*
5	LT.01.double	DR.Rules	RL.expr.double	1	*20 + 20*
6	LT.01.double	DR.Rules ES.finished	RL.expr.plus True	1	*40*

Table 4.1: Step vectors

Table 4.1 shows the step vectors for steps taken. The vector for step 0 was inferred from a message that started the exercise. The indicator DR.created is one of the flags from which the exercise state is inferred, and the exercise now in state BUSY. This was a triple nested double exercise. The last vector contains the finished flag that triggers the rule.

Component	indicator	indicator	value
LT.01.double	ES.objective	EX.double	True
		PRE.plus	Comprehension
	ES.successCriteria	RL.expr.double	1
		RL.expr.plus	1

Table 4.2: Exercise double at start of exercise

Table 4.2 shows the content of the exercise part of the model at the start of an exercises.

Component	indicator	indicator	value
LT.01.double	ES.objective	EX.double	True
		PRE.plus	Comprehension
	ES.successCriteria	RL.expr.double	1
		RL.expr.plus	1
	DR.Rules	RL.expr.double	3
		RL.expr.plus	3
	ES.finished	True	
	DR.created	True	

Table 4.3: Step vectors added to exercise

Table 4.3 show the exercise after all step vectors have been added. The last vector has set the finished flag. The engine will now fire the Succes rule.

Component	indicator	indicator	value
LT.01.double	ES.objective	EX.double	True
		PRE.plus	Comprehension
	ES.successCriteria	RL.expr.double	1
		RL.expr.plus	1
	DR.Rules	RL.expr.double	3
		RL.expr.plus	3
	ES.finished	True	
	DR.created	True	
	ES.success	True	

Table 4.4: State after rule execution of exercises double

Component	indicator	value
ES.student	EX.double	True
	PRE.plus	Comprehension

Table 4.5: Updated student model after exercise double

Table 4.4 and table 4.5 show the outcome of the rule execution. The exercise has the success flag set. It may be displayed as such in the user interface, and it prevents the rules from firing again. The student model was empty and now the objectives have been added. The student has shown that he comprehends the prelude function plus. The student model also indicates that he has executed an exercise of type double.

4.3 Scenario: A missing prerequisite

Component	indicator	indicator	value
LT.11.maxi	ES.prerequisite	EX.double	True
	ES.objective	EX.maxi	True
		PRE.gtr	Comprehension
		SYN.if	Comprehension
	ES.successCriteria	RL.expr.gtr	..

Table 4.6: Prerequisites for exercise maxi

Table 4.6 shows part of the content of an exercise of type maxi. Note that maxi has a prerequisite of EX.double, which the normal student meets after exercise double. The predicates in the exercise recommendation query (section 3.4.4) can be used to determine if the exercise is difficult (prerequisites not met), passed (objectives already met) or the right level. Exercises of type double is now classified as passed, of course the student can take more exercises double if desired. We now recommend exercise maxi, the student meets the prerequisites, but has not yet met the objectives. An exercise of type combined has EX.maxi as prerequisite and is classified as difficult. This way we can guide the student through the curriculum.

If the student would start an exercise without meeting this prerequisite, the user interface may check this and display a message recommending a resource to fill the gap.

4.4 Scenario: A slow student

Component	indicator	indicator	value
LT.11.maxi	DR.Rules	RL.expr.maxi	1
	DR.created	True	
	ES.failure	True	
	ES.finished	True	
	ES.objective	EX.maxi	True
		PRE.gtr	Comprehension
		SYN.if	Comprehension
	ES.prerequisite	EX.double	True
	ES.successCriteria	RL.expr.gtr	1
		RL.expr.if	1
		RL.expr.maxi	1

Table 4.7: Indicators for exercise maxi after error

Component	indicator	value
ES.student	EX.double	True
	PRE.plus	Comprehension

Table 4.8: Updated student model after exercise maxi

Table 4.7 shows the state of the exercise after the exercise was finished and the rules executed. The exercise was finished, but in the DR.Rules vector we see that only rule expr.maxi was executed once. This is not good enough, we have not seen expr.if and expr.gtr. The success criteria are not met and the exercise enters state FAILURE indicated by ES.failure true.

Table 4.8 show the student model. The student completed the double exercise, but it was not updated with the objectives of maxi.

We can not infer what exactly happened. The student may have requested procedural support, and the domain reasoner demonstrated the steps. Another possibility is that the student may have just typed the final answer and skipped steps. A hint may be accounted for by setting a low PiBloom indicator (Knowledge) on a topic. We did not make this refinement here.

4.5 Scenario: A fast student

Component	indicator	indicator	value
LT22.combined	DR.Rules	RL.expr.double	1
		RL.expr.gtr	1
		RL.expr.if	1
		RL.expr.maxi	1
		RL.expr.plus	1
	DR.created	True	
	ES.finished	True	
	ES.objective	CAUS.expressionEvaluation	Comprehension
		EX.combined	True
		PRE.gtr	Comprehension
		PRE.plus	Comprehension
		SYN.if	Comprehension
	ES.prerequisite	EX.maxi	True
	ES.success	True	
	ES.successCriteria	RL.expr.double	1
		RL.expr.gtr	1
		RL.expr.if	1
		RL.expr.maxi	1
		RL.expr.plus	1

Table 4.9: Indicators for exercise combined for a fast student

Component	indicator	value
ES.student	CAUS.expressionEvaluation	Comprehension
	EX.combined	True
	EX.double	True
	EX.maxi	True
	PRE.gtr	Comprehension
	PRE.plus	Comprehension
	SYN.if	Comprehension

Table 4.10: Updated student model after exercise combined

This student started immediately with exercise combined, ignoring any messages on missing prerequisites. Table 4.9 shows the educational state of the exercise after the exercise finished. The combined exercises consist of nested maxi and double expressions. The step vectors are collected in DR.Rules, the student applied all the rules. When the exercise finished, the success rule fired. The state was updated. ES.success was set to True, corresponding to exercise state SUCCESS.

In the student model (table 4.10) we see all objectives added to the student model. However EX.double and EX.maxi have also been set. The fast student rule (section 3.4.3) fired after the success rule. It added the prerequisite EX.maxi. The student now meets the objectives of exercise maxi, so the prerequisites of maxi are added and this added EX.double to the student model. The rule then fired for the double exercise, but this did not add any new information and a fixpoint is reached.

The effect of this is that exercises double and maxi are no longer recommended. If the exercises are still presented somewhere in a menu, the student can take one if he wants.

4.6 Scenario: A misconception

Component	indicator	indicator	value
LT.06.double	ES.objective	EX.double	True
		PRE.plus	Comprehension
	ES.successCriteria	RL.expr.double	1
		RL.expr.plus	1
	DR.Rules	RL.expr.double	1
	DR.Buggy	BUG.buggy.plus	1
	ES.failureCriteria	BUG.buggy.plus	1
	ES.finished	True	
	DR.created	True	
	ES.error	True	
PT.01.fractions	ES.prerequisite	BUG.buggy.plus	1
	ES.objective	FIX.buggy.plus	1
PT.02.fractions	ES.prerequisite	BUG.buggy.plus	1
	ES.objective	FIX.buggy.plus	1

Table 4.11: Indicators for exercise after a misconception

Component	indicator	value
ES.student	BUG.buggy.plus	1

Table 4.12: Updated student model after exercise combined

Certain errors are made very often, and this is an indication that the student has a misconception. An example in our curriculum is working with fractions. An error that is sometimes made is that the numerators and denominators are added, e.g. $\frac{3}{4} + \frac{4}{5} = \frac{7}{9}$. This neutralises our doubling operation, e.g. $\frac{3}{4} + \frac{3}{4} = \frac{6}{8} = \frac{3}{4}$. A domain reasoner can recognise such a mistake with a buggy rewrite rule.

In table 4.11 we see the result. For buggy rules a step vector wil not put the rule execution in DR.Rules but in a vector DR.Buggy. The vector will also contain a finished indicator, as the exercise stops. This vector normally should be empty (bottom). The buggy inference rule now fires, and adds the DR.Buggy content to the student model (table 4.12).

We have anticipated this situation and included some part task exercises to work with fractions. These exercises have the buggy rule BUG.buggy.plus in their prerequisites. The recommendation query will now return the part task practice. We use the same rules for part-task practices and learning tasks.

4.7 Scenario: Variations

The TenSteps methodology recommends variations in exercises. This prevents a student from getting a one sided view or just learn a trick. An airline pilot must make landings in head wind and cross wind conditions. An algebra student with linear equations must make exercises in which alternatively x or y is eliminated.

This places a bit of burden on the exercise designer as we must now keep track of variations. Let us assume we have maxi exercises with either if-syntax or guards as a variation.

Component	indicator	indicator	value
LT.11.maxi.if	ES.objective	EX.maxi1	True
		PRE.gtr	Comprehension
		PRE.if	Comprehension
	ES.prerequisite	EX.double	True
LT.21.maxi.guard	ES.objective	EX.maxi2	True
		PRE.gtr	Comprehension
		PRE.guard	Comprehension
	ES.prerequisite	EX.double	True
LT.33.combined	ES.prerequisite	EX.maxi1	True
	ES.prerequisite	EX.maxi2	True

Table 4.13: Exercise maxi with variations

Table 4.13 shows the prerequisite and objective structure of the maxi variants. When the student has finished double, both variants are recommended. When the student completes LT.11, all other instances of the variant (LT.12, LT.13..) are no longer recommended, and the student is guided to LT.21, the other variant.

The next exercise (LT.33) specifies both variants as a prerequisite.

4.8 Programming tutors

In this chapter we presented scenarios with class 2 exercises. We have not worked out scenario with a class 3 exercises. In this section we describe how domain reasoners for class 3 exercises might interact with the model.

Programming tutors such as Ask-Elle can conduct exercises of class 3. They do not start with a predefined strategy, but with model solutions for which a strategy is derived. The set of solutions is not exhaustive. The student may arrive at a completely different solution that is nevertheless valid.

A domain reasoner such as Ask-Elle can provide procedural support. If needed the tutor can give a hint or a demonstration. Ultimately we want the student to execute the exercise without this procedural support. Using the techniques show before we can guide a student to exercises for which hints are turned off.

A problem is how to update the student model when we cannot count on inferences from rule executions, as the tutor may not have recognised these in some cases. In class 2 exercises we often require a student to follow a certain solution strategy (or SAP). In that case a constraint based update scheme can be used. A domain reasoner parses the solution, and provides a step vector with information on solution conditions. This may include information on a solution correctness, syntax, prelude and language concepts used. For example, the solutions for the mylength exercise in section 2.1.2 listing 5 might provide information on domain concepts.

Domain concept	solution nr
prelude: length	1
syntax: if	2
prelude: null	2
syntax: pattern matching on lists	3
syntax: where	3
concept: strict evaluation	3
concept: recursion	3
concept: higher order function	4
prelude: foldl	4

Table 4.14: Domain concepts in different solutions for listing 5

Table 4.14 shows which concepts from functional programming, syntax and prelude were used in different solutions for the same problem. These concepts can be put in step vectors and used to update the student model, when the exercise is completed successfully.

The updated student model can be used in a task class to steer the student towards solutions. In a side bar we can show a list of domain concepts that we want to be applied. During solving the exercises the concepts used are removed, and the student must clean the bar. If another solution variant is available for an exercise we may refuse a solution that only has familiar concepts. Some creativity from the instructional designer helps to integrate the student model in user experience design.

4.9 Threats to validity

We have some threats to external validity.

4.9.1 External validity

Applicability to other domains

Our approach looks very general and applicable to any domain. We have used examples from the domain of functional programming. We have built another domain reasoner for exercises with linear equations. This is only a limited set of examples.

Step vector calculation

Our examples allowed for an easy translation of rules to step vectors. If the rules correspond to a SAP this is easy. This may not always be so easy, for example, sometimes rules are generated automatically. We do not have a generic procedure for step vector calculation. This may complicate the design or create extra work for the courseware author.

Retrofit

To retrofit the student model on an existing domain reasoner might prove difficult if the educational context is missing. Exercises must have a well defined SAP, prerequisites, objectives and level of difficulty. An extra iteration may be needed and redesign of exercises and solutions. In the mylength example in Section 2.1.2 we showed multiple solutions. Depending on exercise goals one or another solution might be preferable.

Chapter 5

Conclusions

In this chapter we will evaluate how the research questions have been answered, and draw some conclusions. The section on future work describes some new questions and opportunities that arise from the rule system we described.

5.1 Research questions

The objective of this research is to find answers to the following question:

How can a student model based on the TenSteps methodology be created and used in Ask-Elle for inner and outer loop behaviour ?

This question can be divided in the following subquestions.

Q1. How can a student model be used to diagnose learning problems such as missing prerequisites and forgotten or misunderstood topics?

Answer
Missing prerequisites can be detected at the beginning of the exercise using predicates discussed in section 3.4.4. This only means that the student model has no evidence that the student masters the concept. The student may have acquired the prerequisite knowledge outside the ITS. A warning message is displayed, but student may continu the exercise.

A buggy rule may fire that indicates a problem with a prerequisite. If the student model indicates that the student in fact meets the prerequisites, we conclude the item is forgotten and otherwise it is a missing prerequisite. When a buggy rule fires that is not related to a prerequisite, it is a misconception that must be addressed in this module.

Q2. How can the student model be used to choose between possible interventions? Typical interventions are: reminding, persuasion, teaching and remediation (VanLehn, 2016).

Answer
Persuasion is used for variations in exercises. In section 4.7 we showed how a student can be persuaded to a specific variation of an exercise.

In case of a problem, the intervention depends on the diagnosis. If an item is forgotten, we can show a pointer to supportive information to remind the student. A missing prerequisite needs teaching, the student is referred to the prerequisite module. In case of misconception some remediation can be offered by referring the student to a part-task practice.

Q3. Completing exercises provides evidence of student mastery. In the TenSteps methodology the amount of guidance needed is an important indicator of student mastery. If different solutions are possible then each solution may provide evidence of mastery of different concepts.
How can the inner loop behavior be used for knowledge tracing in the student model?

Answer
If procedural support is given or an error made, the student part of the model is not updated. The exercises is left in the state INCOMPLETE or ERROR. This may be refined further (section 5.3.1). In the examples we showed how the step vector is used to update the DR.rules vector for this purpose.

There are many other kinds of updates possible with a step vector. In case a hint is given, the student model can be update directly without any rules. For example, the PiBloom level of knowledge can be set on the hinted topic. If the hint is requested again the student we may encouraged to try to solve it himself.

A step vector can add some values to the objectives vector. In case of successful completion these will be included in the student model, as we showed in section 4.7 with the syntax variations.

Q4. How can the outer loop control the scheduling of learning tasks and task classes following the TenSteps methodology?
In the TenSteps methodology the student can progress to the next task class when no procedural guidance is needed anymore. Variability of practice is important in the TenSteps methodology for a good transfer of learning.

Answer
We have shown that we only update the model after successful completion of exercises. This contributes to certainty that the student masters the exercise.

In the outer loop examples we showed that exercise selection can be refined as desired for progressing levels of difficulty. We did the same for variability in the inner

loop examples. This mechanism may also be used to implement macroadaptation (VanLehn, 2006).

For task classes a rule may be defined similar to the success rule for learning tasks. Taks classes can only be finished successfully or remain unfinished.

Q5. What is the impact of the student model on content authoring?
Exercises and model solutions must be linked to domain concepts and learning objectives. This makes the authoring of artifacts more complex.

Answer
Our model is declarative with regard to educational properties. It is not necessary to write extra rules or algorithms to add new exercises.

Domain concepts and levels of learning can be used in the student model and reused as prerequisites and objectives. This effort is scalable in the sense that it may require more work, but will result in a more educationally sound solution.

Q6. How can we build a lightweight knowledge representation and knowledge inference mechanism?

Answer
Many ITSs use algorithms such as Bayesian logic, fuzzy logic and machine learning (Chrysafiadi and Virvou, 2013). The mechanism we described is lightweight compared with these solutions. Our prototype is only three Haskell modules with about 500 LOC to define the progress indicators, the rules for TenSteps methodology. It requires little parameterisation, and is driven by domain concepts and levels of learning.

Even in a large curriculum we do not anticipate performance problems. Our engine loops over all task classes and learning tasks. The engine can be optimised per rule to just select the current task or task class.

We have no functionality for evaluation and prediction of student performance. Other approaches have better support in this area.

5.2 Conclusions

Development of an ITS is a risky and complex undertaking. A lightweight approach facilitates fast and easy creation of a student model early in the lifecycle. Following the TenSteps methodology and designing a student model adds an educational dimension to exercise creation.

Some intelligent technology is required to reason about student progress and determine interventions and recommendations. We have used the blueprints from the TenSteps methodology to structure our educational and student model. Partial orderings and lattices are mathematical structures, that are useful to reason about curricula.

The knowledge space theory, for example, uses partial orderings, and so do we in our model.

We have defined a model and a rule system with the following properties.

- Our rule system can be used for outer loop navigation, and for inner loop interventions.
- We only had to define 5 rules for navigation and interventions. The model and the rules follow the blueprints of the TenSteps methodology.
- The rules work well with class 1 and class 2 exercises.
- Working with class 3 exercises may require additional rules to reason about constraints. Our system can be expanded in various ways (see Section 5.3).
- The rules are not specific for any knowledge domain.
- Exercises, navigation and interventions can be defined declaratively. Authors only have to specify properties.
- The system is easy to understand for both students and instructors. The system advices, but does not take over responsibilities.

This approach is suitable for a light weight student model. It does not present challenges in computational complexity. It requires limited resources, and can be useful in the early phases of the lifecycle of an ITS. Other student modelling approaches (Corbett and Anderson, 1995) require more work to set up and parameterise. These models offer more functionality than our approach. For example, our model has no facilities to predict student performance in exams.

5.3 Future work

We have explored some of the possibilities of a lattice based student model. We have build a only a proof of concept, however the experience indicates that this is a promising direction. We feel that we have only shown the top of the iceberg and that this approach has many more possibilities. In this section we show that there are plenty of ideas, to encourage futher research in this direction.

5.3.1 Expansions

Refinements for procedural support

Our model may be a little coarse. We have no separate state for an abandoned exercise, or an exercise where procedural support was given. We do not keep track of the amount of hints and procedural support given. Sometimes a student needs some encouragement to try it himself, and maybe the system should restrict the amount of support. How can this be further refined and give more refined amount of procedural support?

Other educational technologies

We have tested the model with domain reasoners from the IDEAS project. Maybe it is possible to use the model with other educational technologies. If a step vector can be derived from student interactions then the model can be updated. The model will then contain more information on the student cognitive state and learning preferences. For example an intake form with multiple choice questions may be used to update the student model. The knowledge space theory reasoning capabilities can be used to select questions based on previous answers.

Application in other domains

Our model in not very domain specific. It may be applied in other domains.

Class 3 domain reasoners

Further experiments in different domains are required to draw conclusions on domain reasoners for class 3 problems in combination with our model. There may even be differences per domain e.g. SQL, imperative programming and functional programming. What are the design aspects of domain reasoners for class 3 problems in combination with the student model? Domain reasoner need to be adapted to generate step vectors to update the model.

5.3.2 Improvements of the underlying rule engine

We have tried to implement a student model in OWL/SWRL. OWL/SWRL is based on description logics. Description logics allow lightweight and monotonic reasoning, where new facts can asserted. This is what we need in a student model where information is only added. Theoretically this should work fine together, but we found it difficult to reason with time dimensions in OWL/SWRL. More research is required to integrate temporal logic, description logic and lattice theory, to build a more generic rule engine.

Accumulating results over time

Step vectors are called asserted facts, output from rules are called inferred facts. We obtain a series of stepvectors $\{SV_1,\ SV_2 ...\ SV_n\}$ from the user dialog. The first model M_0 is an asserted model obtained from the courseware author. M_0 contains the definitions of learning tasks and an empty student model. The rule engine is a function re that calculates the next version of the model from the current version by joining rule results until a fixpoint is reached. In our prototype we generate a series of models $M_n = re(M_{n-1} <+> SV_n)$, that contain asserted and inferred facts.

Another approach would be to first calculate a series of models with only asserted facts: $\{AM_1,\ AM_2 ...\ AM_n\}$, where $AM_n = AM_{n-1} <+> SV_n$ and $AM_0 = M_0$. The rule engine is then applied to these models: $M_n = re(AM_n)$. The inferred facts are not

carried over from step to step, and this relaxes the restriction of the model increasing monotonic.

Monotonic reasoning

We experimented with OWL/SWRL. This is a lightweight inferencing system, based on description logic, and supports open world reasoning. This system is not very sensitive to rule execution order, and avoids conflicting rules with monotonous reasoning. We are monotonous in the sense of the model increasing monotonic in partial ordering. We have lost some of the advantages of monotonous reasoning. In our model many rules may influence each other. We avoided this problem by defining rules on predicate combinations that exclude each other. This is in general not the case and may introduce complexity such as setting priorities on rules. More research is needed how to improve on this point.

Other rule systems

Our rule engines is dedicated to support the TenSteps methodology. Perhaps it is possible to integrate the lattice reasoning in existing rule engines to build a more generic solution.

Explanations

A student may disagree or wonder how the system concluded that he understands e.g. recursion. Sometimes an explanation facility is needed to explain which rules and assertions have resulted in certain facts. Our system does not have such a mechanism.

5.3.3 Lifecycle and migration

When an ITS matures, more functionality will be required.

Metrics and performance predictions

We have not explored any possibility to use the model to evaluate student knowledge or to make predictions on performance. Perhaps querying learning curves can provide metrics on student performance.

Counters for individual learning curves.

Some students may need more exercises than others and therefore the educational model may be parameterised with counters per individual. It may be useful to be able to make rules with conclusions that are added to the model (<+>) instead of joined ($\vee$). If such rules keep firing, a fixpoint is never reached and the engine enters an infinite loop. This can be solved by defining one-shot rules.

Multi dimensional indicators

We used the PiBloom indicator to state the level of learning. Such a value can probably be augmented with extra information indicating how often or with what probability this is asserted.

Migration

We have developed the model as a starting point in the early lifecycle of an ITS. A model based on Bayesian statistics has many advantages in a large scale ITS. So when the ITS matures, such capabilities must be added.

Out model supports explicit definitions for prerequisite relations, in terms of skills or prerequisite relations between tasks or task classes. Maybe this prerequisite model can be used to define a bayesian belief network. In that case statistical update rules may be defined to update the student model.

Glossary

ACT-R (Adaptive Character of Thought). A theory oflearning and cognition based on a theory of the mind containing a large number of rules.

Ask-Elle A tutor based on an IDEAS domain reasoner for functional programming in Haskell.

CAI Computer Aided Instruction. A form of electronic learning.

CBT Computer Based Training, synonym for CAI.

Constraint based tutor An intelligent tutor that gives feedback based on constraints on the solution. For example an SQL tutor will require that a solution is syntactically correct and yield the right results.

Domain reasoner A domain reasoner can reason about student solutions in a particular domain. It can diagnose errors, evaluate if the student is still on the right track, or advice on a next step to take.

HEE Haskell Expression Evaluator, a tutor that conducts exercises where functional programs are evaluated as expressions.

Inner loop The innerloop uses intelligent technology to coach a student through an exercise. It works on the task step level.

ITS An Intelligent Tutor System uses intelligent technology to give feedback and hints during exercise execution.

Knowlegde space theory A psychometric method for question selection based on prerequisite relations between items.

LOC Lines of code. Different interpretations are possible with regard to empty lines and comment lines.

Lucky guess A student does not know the correct answer, but picked the right solution anyway

macroadaptation An outer loop mechanism where tasks are selected that contain only one or two new knowledge components.

Model tracing tutor A tutor that gives feedback by tracing student progress against an expert solution. The tutor recognises not only mistakes, but also deviations from the solution strategy.

Outer loop The outer loop is responsible for selecting exercises and learning tasks. It can vary from a menu where the student can choose any exercise to an advanced recommendation system.

SAP A Systematic Approach to Problem Solving.

Slip A student made an error although he understands the correct solution.

Student model A part in the ITS where information about the student, such as learning progress and preferences, is maintained.

TenSteps methodology The Ten Steps/Four components instructional design methodology

Bibliography

Ayesha Ameen, Khaleel Ur Rahman Khan, and B Padmaja Rani. Ontological student profile. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, pages 466–471, New York, NY, USA, 2012. ACM.

John R. Anderson. ACT: a simple theory of cognition. *American Psychologist*, 51(4), 1996.

John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. Skill acquisition and the LISP tutor. *Cognitive Science*, 13(4):467 – 505, 1989.

Franz Baader, Diego Calvanese, Deborah L. McGuiness, Danielle Nardi, and Peter F Patel-Schneider. *The description logic handbook*. Cambridge University Press, 2007.

Supriya Bhalerao, Anagha Ranade, and Ashok D.B. Vaidya. Bloom's taxonomy reiterates pramana. *Journal of Ayurveda and Integrative Medicine*, 8(1):56 – 57, 2017.

P. L. Brusilovsky and E. Millán. The adaptive web. pages 3–53. Springer-Verlag, Berlin, Heidelberg, 2007.

P.L. Brusilovsky. The construction and application of student models in intelligent tutoring systems. *Journal of Computer and Systems Sciences International*, 32(1) (1994), 1992.

P.L. Brusilovsky. Student model centered architecture for intelligent learning environments. In *Fourth international conference on User Modeling*, 1994.

Konstantina Chrysafiadi and Maria Virvou. Student modeling approaches: A literature review for the last decade. *Expert Systems with Applications*, Volume 40, 2013.

Albert T. Corbett and John R. Anderson. Knowledge tracing: Modeling the Acquisition of Procedural Knowledge. *User Modeling and User-Adapted Interaction*, 4:253–278, 1995.

Jean-Paul Doignon and Jean-Claude Falmagne. *Knowledge spaces and learning spaces*, volume 1 of *Cambridge Handbooks in Psychology*, pages 274–321. Cambridge University Press, 2016.

Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Properties of exercise strategies. In *First International Workshop on Strategies in Rewriting, Proving and Programming (IWS2010)*, volume EPTCS 44, pages 21–34, 12 2010.

Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L. Thomas van Binsbergen. Ask-Elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2017.

Abigail S Gertner and Kurt VanLehn. Andes: A coached problem solving environment for physics. In *International conference on intelligent tutoring systems*, pages 133–142. Springer, 2000.

Ioannis Hatzilygeroudis and Jim Prentzas. *Knowledge Representation Requirements for Intelligent Tutoring Systems*, pages 87–97. Springer Berlin Heidelberg, 2004.

Bastiaan Heeren and Johan Jeuring. Feedback services for stepwise exercises. *Science of Computer Programming*, 88:110 – 129, 2014.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, second edition edition, 2016.

Zoran Jeremić and Vladan Devedžić. *Student Modeling in Design Pattern ITS*, pages 299–305. Springer Berlin Heidelberg, 2004.

Johan Jeuring, L. Thomas van Binsbergen, Alex Gerdes, and Bastiaan Heeren. Model solutions and properties for diagnosing student programs in Ask-Elle. In *Proceedings of the Computer Science Education Research Conference*, CSERC '14, pages 31–40, New York, NY, USA, 2014. ACM.

Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. Strategy-based feedback in a programming tutor. In *Proceedings of the Computer Science Education Research Conference*, CSERC '14, pages 43–54, New York, NY, USA, 2014. ACM.

Kenneth R. Koedinger, John R. Anderson, William H. Hadley, and Mary A. Mark. Intelligent tutoring goes to school in the big city. *International Journal on Artificial Intelligence in Education*, 8:30–43, 1997.

Philippe Kruchten. Lifelong learning for lifelong employment. *IEEE Software*, July/August 2015.

Nguyen-Thinh Le, F. Loll, and N. Pinkwart. Operationalizing the continuum between well-defined and ill-defined problems for educational technology. *IEEE Transactions on Learning Technologies*, 6(3):258–270, 2013.

Tim Olmer, Bastiaan Heeren, and Johan Jeuring. Evaluating haskell expressions in a tutoring environment. *Trends in Functional Programming in Education*, EPTCS 170: 50–66, 2014.

Peter Reimann, Michael Kickmeier-Rust, and Dietrich Albert. Problem solving learning environments and assessment: A knowledge space theory approach. *Computers and Education*, 64, 2013.

Garrett Birkhoff Saunders MacLane. *Algebra*. AMS Chelsea publishing, third edition, 1988.

Jeroen J. G. van Merriënboer and Paul A. Kirschner. *Ten steps to complex learning*. Routledge, New York, second edition, 2013.

Kurt VanLehn. *Foundations of Intelligent Tutoring System*, chapter Student Modelling. Lawrence Erlbaum Associate Publishers, 1988.

Kurt VanLehn. Cognitive skill acquisition. *Annu. Rev. Psychol.*, Vol 47:pages 513:539, 1996.

Kurt VanLehn. The behavior of tutoring systems. *Int. J. Artif. Intell. Ed.*, 16(3): 227–265, 2006.

Kurt VanLehn. Regulative loops, step loops and task loops. *International Journal on Artificial Intelligence in Education*, 26:107–112, 2016.

Kurt VanLehn, Kenneth R. Koedinger, Alida Skogsholm, Adaeze Nwaigwe, Robert G. M. Hausmann, Anders Weinstein, and Benjamin Billings. *What's in a Step? Toward General, Abstract Representations of Tutoring System Log Data*, pages 455–459. Springer Berlin Heidelberg, 2007.

Beverly Park Woolf. *Building Intelligent Interactive Tutors*. Morgan Kaufmann, Burlington, Massachusets, second edition, 2010.