Aspects of Feedback in Intelligent Tutoring Systems for Modeling Education

Proefschrift

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. mr. A. Oskamp
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op vrijdag 13 september 2013 te Heerlen
om 14.00 uur precies

door

Henricus Jeroen Maria Passier

geboren op 18 augustus 1962 te Hilversum

Promotor

Prof. dr. A. Bijlsma	Open Universiteit

Overige leden beoordelingscommissie

Prof. dr. E. Barendsen	Radboud Universiteit
Prof. dr. M.C.J.D. van Eekelen	Open Universiteit Radboud Universiteit
Prof. dr. ir. S.M.M. Joosten	Open Universiteit
Dr. R. Kuiper	Technische Universiteit Eindhoven
Dr. B.J. Heeren	Open Universiteit

Cover by Monique Vossen, Visuele Communicatie, Open Universiteit
Thesis printed by Canon business services, Heerlen

ISBN: 978-94-91825-20-0

Copyright © 2013 by H.J.M. Passier

Contents

1 Introduction **1**
1.1 Research context . . . 1
1.2 Examples . . . 3
1.3 Modeling . . . 5
1.4 Feedback . . . 9
1.5 Restriction on models . . . 11
1.6 Examples extended . . . 13
1.7 Summary . . . 16
1.8 This thesis . . . 17
1.9 Origin of chapters . . . 18

2 A framework for practicing modeling **21**
2.1 Introduction . . . 21
2.2 Main concepts and functions . . . 24
2.3 The framework . . . 36
2.4 Additional concepts . . . 40
2.5 Related work on modeling ITSs . . . 50
2.5.1 Model-based . . . 51
2.5.2 Constraint-based . . . 54
2.5.3 Authoring . . . 56
2.6 Research questions . . . 58
2.7 Haskell preliminaries . . . 58

3 Feedback in an ITS for solving equations **61**
3.1 Introduction . . . 61
3.2 The feedback framework . . . 66
3.3 Syntax analysis . . . 69
3.4 Rewriting terms . . . 70
3.5 Progression and indicators . . . 73
3.6 Hints . . . 74

3.7 A general tool 75
3.8 Related work 76
3.9 Conclusion 80

4 Feedback in authoring tools 83
4.1 Introduction 83
4.2 Schemata and schema representations 85
4.3 Schema analysis to detect authoring problems 87
4.3.1 Data structures and definitions 88
4.3.2 Solving authoring problems with schema analysis . . . 91
4.4 Related work 96
4.5 Conclusion 97

5 Supporting several model languages 99
5.1 Introduction 99
5.2 UML-Class diagrams 100
5.2.1 An example 100
5.2.2 Syntax 101
5.2.3 Semantics 102
5.2.4 Types of errors 102
5.3 The framework 105
5.3.1 The central model 106
5.3.2 Translations 108
5.3.3 Analysis functions 112
5.4 Related work 113
5.5 Conclusions 116

6 From ill-defined to well-defined tasks 117
6.1 Introduction 117
6.2 DTDs and Regular Expressions 119
6.2.1 Syntax 119
6.2.2 Language 120
6.2.3 Rewrite rules 121
6.3 Removing non-determinism 122
6.3.1 Strategy for removing non-determinism 123
6.3.2 Normal form for content models 125
6.4 Precise content models 126
6.4.1 Strategy for precise content models 127
6.4.2 Using the strategy 127
6.5 Correct content models 128

6.5.1 Strategy for correct content models 128
6.5.2 Using the strategy 129
6.6 Experiment and validation 130
6.6.1 Results 130
6.6.2 Discussion 131
6.7 Related Work 132
6.8 Conclusions 132

7 Epilogue and future work 133
7.1 Summary 133
7.2 Putting things into perspective 135
7.3 Future work 139

Bibliography 141

Samenvatting 157

Dankwoord 169

Curriculum Vitae 171

Chapter 1

Introduction

1.1 Research context

The research in this thesis is about automatic feedback generation by intelligent tutoring systems during modeling as an activity in courses that are part of a bachelor computer science curriculum.

Modeling plays a crucial role in computer science. Examples of modeling activities include Object-Oriented (OO) modeling (for example domain modeling, use case modeling, communication modeling, and design modeling) using the notation Unified Model Language (UML) [21, 85], database modeling using the Entity Relationship (ER) model notation [30], process modeling using Petri nets [1, 127], and string modeling using regular expressions [66]. Courses in the bachelor curriculum Informatics at the Open Universiteit (OU), in which these modeling activities play a role, are [81, 136, 157, 158, 160].

Modeling is a difficult and creative activity. Students, when learning modeling, can make many mistakes, syntactic as well as semantic. To optimize the learning process, immediate feedback from a lecturer is essential [113]. This is a problem at the OU, because due to our emphasis on distance education the number of contact hours is very limited.

In distance education, students study their study material mostly at home and individually. The material includes both the learning content and supporting components such as study hints, exercises with detailed answers, and summaries. Although the material is developed to be studied independently, limited additional guidance is organized in the form of lectures and discussion groups in which lecturers and students can meet each other.

There are two types of lectures, namely face to face and online. In a face to face lecture, lecturer and students meet each other physically in a room.

In an online lecture the lecturer and students meet each other in a virtual class room. A virtual class room is a web- or software-based online learning environment where students participate in synchronous instruction. The number of lectures for a course of 4.3 ECTS (nominally 120 study hours) is usually five face-to-face lectures of three hours or about eight online lectures of one and a half hours. The current trend is an increase in online meetings at the cost of face to face lectures. It should be clear that, in comparison with regular universities, the number of contact hours by means of lectures is very limited and thus so are the possibilities of giving immediate feedback.

Besides lectures, each course is supported by a discussion group. A discussion group is a web-based or an Usenet-based service in which students and lecturers can communicate with each other. Within such a group, students can ask questions so that fellow students and lecturers can answer them and discuss subject matter. One drawback of this medium is its asynchronous character which causes feedback to be delayed. Furthermore, it is definitely not the intention to discuss all students' solutions on exercises part of the study material, because this would be very labor intensive. It is as Sara Guri-Rosenblit [56] stated: 'The lack of direct teacher and student-student communication has been the Achilles heel of distance education for centuries'.

In this context, it is obviously useful to investigate the possibilities of an e-learning system in which students can learn to develop models by practicing and for which the system produces immediate feedback on the modeling steps and subsequent artifacts made by the student.

It should be emphasized that the focus in this research is on practicing: i.e. students first receive information about the modeling skill by reading books, studying examples, and possibly by attending online lectures. After that, a phase of practicing takes place during which a student can complete a number of exercises and receives feedback about, for example, the steps taken and the quality of the resulting model. This means that the e-learning system we will develop is assumed not to be used for learning theory about modeling.

In the next sections, we will explain the problem of the lack of immediate feedback on individual modeling activities in more detail. We will present two examples, describe what modeling means in the context of computer science, what feedback is and why it plays a crucial role in learning modeling, and what types of models will have our attention in this research. In the next chapter, we will describe the ideas of an intelligent tutor system, a type of e-learning system that automatically generates feedback during modeling activities.

1.2 Examples

To explain what we mean by modeling and feedback, we firstly present two examples. The first example is about modeling an XML content model [158]. The second example is about modeling a domain model [81, 136].

Example one: Modeling an XML content model. In a course about XML, students learn about the schema language Document Type Definition (DTD) [109]. Roughly, a schema written in the DTD language lists a number of element declarations where each element declaration consists of the name of the element declared and a content model. The content model specifies the names of the child elements that may occur and in which order. For example, in

```
<!ELEMENT book (title, author+, chapter+)>
```

an element with name `book` is defined. The content model is `(title, author+, chapter+)`, which means that each element `book` must have an `title` element, followed by one or more `author` elements, and one or more `chapter` elements.

In an exercise, the studnet is asked to define a content model for element `rec` based on the following example XML document:

```
<recs>
  <rec>
    <a/><b/>
  </rec>
  <rec>
    <a/><b/><a/><b/>
  </rec>
  <rec>
    <a/><b/><c/>
  </rec>
  <rec/>
</recs>
```

The answer given at the end of the chapter is: 'A correct content model is `(a, b, (a, b | c)?)?`. Other solutions are possible.'.

It is important to note that in courses and books about XML, as far as we know, no systematic way for modeling an XML content model is presented[1]. As a result, students solve these problems by their own trial-and-error methods resulting in correct but also incorrect models mostly reached

[1]In one book [109], it is observed that element content can be written using a variation of the regular expression notation. However, it does not explain how to reach a 'good' model.

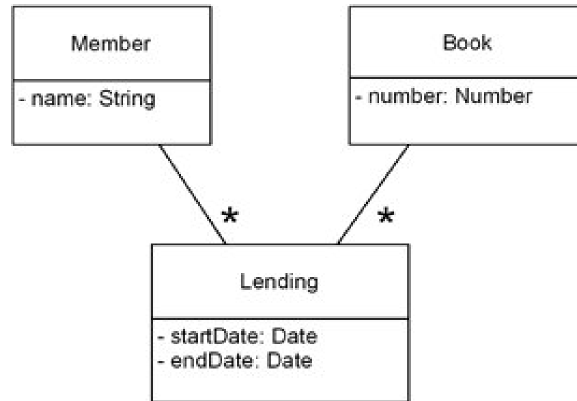


Figure 1.1: A domain model for lending books from a library

in an inefficient way[2].

Suppose a student produced the content model `(a,(b,(a, b | c)?))?`. A relevant question the student could ask is: 'Is this correct too?'. Another interesting question the student could ask is: 'Have I reached the solution in an efficient way?'. The answer `(a, b, (a, b | c)?)?` does not answer such questions. In other words, what the student misses here is a recipe consisting of certain steps which should be performed in a particular order, immediate feedback at the level of these steps and the resulting content model, and a way of comparing his or her resulting model to the answer's model.

Example two: Modeling a domain diagram. The second example is about modeling a domain model using the UML class diagram notation [85]. A domain model consists of classes representing concepts in the domain of interest, attributes of these classes, and associations between these classes.

The exercise asks the student to make a domain model of the following situation: 'A library lends books. We speak about a lending when one book is borrowed by one member and concerns a certain period of time. A member can borrow several books.' The answer given is: 'Figure 1.1 shows a possible diagram'.

Suppose a student produced the domain model showed in figure 1.2. Again, the student cannot find an answer on the question 'Is my diagram correct too?'.

[2]Recently, we have observed this by a number of think-aloud sessions during which students modeled some content models.

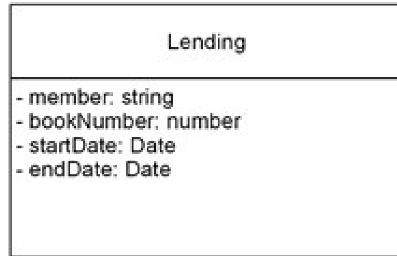


Figure 1.2: Another domain model of lending books from a library

1.3 Modeling

The previous section presented two examples of models. This section gives an overview of what modeling is and what types of models exist in the context of designing an information system.

Definitions. Generally, many definitions exist of what a model is and what modeling is [65, 140]. An elementary universal description is that 'A *model* is a replication of a real world entity that requires some simplifying assumptions and may be physical or conceptual' [65]. *Modeling* then is the process of constructing such a model. In our context, we can use the definition of Sommerville, i.e. '*Modeling* means developing abstract models of an information system, with each *model* presenting a different view or perspective of that system' [140]. Sommerville talks explicitly about model*s* because an information system is such a complex system that more than one model is often needed to specify sufficiently the system to be developed, and each model describes the system from a specific perspective.

Notation. A model represents a system and uses some kind of *notation*. There are several classifications of these languages [65, 140]. In the context of information systems, we can mainly distinguish four types of languages, namely graphical versus textual languages and formal versus informal languages. A *graphical language* uses a diagram technique with for example named symbols representing concepts, lines representing relationships, and various other graphical notations to represent constraints. A *textual language* uses a natural or mathematical languages. A *formal language*, comes with a formal set of rules which defines its syntax and semantics precisely, whereas an *informal language* misses these formal set of rules. Some examples of each category are:

- *Formal - graphical*: Petri nets, UML class diagrams and ER-diagrams
- *Informal - graphical*: Structured Analysis and Design Technique (SADT)[3].
- *Formal - textual*: DTDs, regular expressions and proposition logic
- *Informal - textual*: UML use case descriptions

Semantics. Informally, the *semantics* of a model specifies a mapping between 'the symbols used and the way they are structured' in a model and 'what these structured symbols mean'.

For example, in figure 1.1 the model consists of two different symbols, namely named rectangles and lines between these rectangles. The rectangles are interpreted as classes, where each class is a set of objects. The lines between the rectangles are interpreted as associations, where each association expresses a relationship between the objects of those classes. As a result, each (syntactically correct) UML class diagram has a certain meaning. In case of figure 1.1, there are three classes (Member, Lending and Book) and two relationships (one relationship between the objects from the classes Member and Lending, and one relationship between the objects from the classes Lending and Book).

The semantics of a model can be correct or incorrect. A model is semantically incorrect, if the semantics of that model deviates from the semantic truths concerned in the domain of interest. For example, we can make a UML class diagram representing that 'a car has one wheel'. The semantics of this model is 'a car has one wheel', but in our real world the semantics of this model is incorrect.

Categories of models. There exist several ways of categorizing these models [65, 82, 94, 140]. Sommerville distinguishes the following types of models [140]:

- *Context models* – This type of models represents the system from an external perspective and helps in deciding on the system boundaries, i.e. which functions should be included in the system and which are provided by the environment of the system. Context models might describe the organization of the context in terms of the concepts involved and/or the (automated) processes in which the system will be used. An example of a context model is the context model as part of

[3] See: http://www.cs.toronto.edu/ jm/2507S/Notes04/SADT.pdf

Jackson Software Development [68]. A UML activity diagram and a UML use case diagram can be used as context models too.

- *Interaction models* – This type of models represents the interactions between a user and the system, between other systems and the system being developed, and between the components of the system. Examples of interaction models are the use case model using the UML use case model notation and the sequence diagram to model the interaction between components of the system.

- *Structural models* – This type of models represents how the system is organized and the structure of the data the system processes. Structural models can describe the static structure as well as the dynamic structure of a system. The static structure of a system describes the components of the system and the static relation between these components. The dynamic structure describes the organization of the system in terms of a set of interacting threads. An example of a structural model is the class model using the UML class diagram notation.

- *Behavioral models* – This type of models represents the dynamic behavior of a system and how the system responds to events. A stimulus could be some data that arrives or some event that happens triggering the system to process. In the first case we talk about data driven models, in the second case we talk about event driven models. An example of a data driven model is the data-flow diagram (DFD), an example of an event driven model is a state diagram using the UML state diagram notation.

Another categorization is distinguishing between analysis models and design models.

- *Analysis models* – We can characterize analysis as specifying the *what*: what is the system supposed to do. Analysis designates some kind of understanding of a problem or situation and captures the requirements without prematurely adopting implementation decisions, i.e. omitting technology dependent details and using concepts solely drawn from the problem domain [49, 78].

- *Design models* – We can characterize design as specifying the *how*: how will it do it. Design is related to the creation of a solution for the analyzed problem and must define a software solution that effectively and efficiently satisfies the requirements specified in analysis.

> In doing this, the design model will often incorporate new artefacts (new concepts, operations, etcetera) and it can take into account the concrete technological platform on which the software system is to be built. Two examples are the controller class in a design model as access point for the user interface classes and an extra association between two classes to make a data retrieval more efficient [85]. In fact, the design has to provide a creative solution for the problem specified in the analysis and takes into account, for example, non-functional requirements such as performance, reuse, maintainability, etcetera [67, 78]. This creative aspect of design models makes it generally more difficult to give meaningful feedback during the creation of a design model as opposed to analysis models.

Modeling and model transformation. During modeling, first an abstract model is made which presents a particular view or perspective of a system. After that, subsequent stages can follow in which the model can be further optimized, simplified, etcetera. These optimizations and simplifications are examples of model transformations. By *model transformation*, one or more target models are generated from one or more source models according to a transformation definition, where each transformation definition is a set of rules that together describe how the transformation is performed [95]. We can distinguish between endogenous versus exogenous transformations and horizontal versus vertical transformations [95]. *Endogenous transformations* are transformations between models expressed in the same language. Examples are model optimization (improve certain operational qualities while preserving the semantics of the model), refactoring (change the internal structure of a model to improve certain qualities without changing its behaviour), and simplification (translating certain syntactic constructs into simpler ones). *Exogenous transformations* are transformations between models expressed in different languages. An example of an exogenous transformation is code generation, where, for example, a design model is translated into source code. Another example is the transformation of some requirements into a first design model. A *horizontal transformation* is a transformation where the source and the target models reside at the same abstraction level. Refactoring is a typical example of this type of transformation. A *vertical transformation* is a transformation where the source and the target models reside at different abstraction levels. An example is model refinement where a model is gradually refined into a full-fledged model ready for implementation by adding more and more concrete details.

Stategies and rules of thumb. Modeling is constructive in nature. A model is often developed stepwise following some phases, such as by identifying the aim of the model, exploring the problem domain, building the model using some rules according to a strategy or some rules of thumb, refactoring and optimizing the model, and finally testing whether the model is acceptable or not. For some models there exists a *strategy* (also procedure or recipe) of how to develop such a model. A strategy describes how basic steps may be combined to solve a particular problem. An example of a model that can be developed according to a strategy is a precise XML content model. We will show an example of this in section 1.6. For other types of models, there are only some *rules of thumb*. A rule of thumb is a principle that is not intended to be strictly accurate or reliable. An example of a model for which only some rules of thumb exist is a domain model. We will show an example of this in section 1.6.

1.4 Feedback

Feedback is as crucial in learning a complex task such as modeling is. Feedback is used in many learning paradigms. It is an accepted psychological principle that one of the essential elements needed for effective learning is feedback [113].

There are many definitions of feedback [28, 34, 132, 150]. Examples of definitions are: 'Feedback is information presented that allows comparison between an actual outcome and a desired outcome' [150] and 'Feedback allows the comparison of an actual performance with some set standard of performance' [75].

Both definitions emphasize the fact that there must be at least one 'desired outcome' or 'standard of performance'. In our context, this means that there is at least a standard model to compare with the student's final model in cases we want to give feedback on the level of the final model. In case we want to give feedback on the level of the student's steps in reaching a final model, we need to know a strategy, which steps in which order are allowed. This is an important issue, because we can only give feedback on models or parts of models for which such a standard model and/or strategy exists.

The best timing of feedback, immediate versus delayed, is still an open question [45, 113]. It seems there exists an agreement that *immediate feedback* produces a better effect during initial stages of learning, i.e. immediate feedback guides the learner and results in superior initial performance as the time required for mastery, whereas *delayed feedback* produces better effect during later stages of teaching, i.e. delayed feedback fosters the development

of secondary skills such as error detection and self-correction [58, 113].

Merriënboer and Kirschner distinguish between cognitive and corrective feedback [96]. *Cognitive feedback* allows the learner to reflect on the quality of the found solutions or the quality of the problem solving process of non-recurrent skills. The main function of cognitive feedback is to foster reflection in the receiver's mind. *Corrective feedback* gives learners immediate information on the quality of the performance of recurrent skills. The main function of corrective feedback is to correct errors.

Mitrovic [105] describes, besides corrective feedback, the importance of positive feedback. *Positive feedback* reduces student uncertainty about tentative but correct steps. An example of positive feedback is: 'Yes, that was a correct step'. Positive feedback is useful when a student is uncertain but nevertheless happens to perform the right step. Positive feedback should be immediately given after a step in the solution process.

We did not find any literature about feedback during modeling activities outside the context of an e-learning system. Literature about feedback during modeling activities in the context of an e-learning system will be discussed in the next chapter. Modeling is at least an important part of problem solving [22, 143, 147, 148, 151], If we consider modeling as problem solving, we can find literature.

Modeling, or more generally problem solving, consists of several stages [137]. These include identifying a goal to reach, exploring the problem domain, selecting and applying multiple rules in some order to reach a solution, and testing whether the solution is acceptable or not. During this process, a learner may use declarative knowledge and cognitive strategies within that domain, and combine relational as well as procedural rules [48]. In their book *Ten steps to complex learning*, Merriënboer and Kirschner talk about schema-based processes, where knowledge is used in the form of cognitive schemata that can be interpreted so as to be able to reason about the problem domain (i.e. mental models) and to guide the problem solving process (i.e. cognitive strategies).

Feedback during problem solving activities must help the learner in finding and following the right strategy or schema, and detection of knowledge gaps. Initial feedback may be in the form of hints, may include the appropriateness of selected solution paths, and correctness of steps. Later, as learners transition from novice to expert, feedback should be more about the efficiency of problem solving [137].

1.5 Restriction on models

The distinction between analysis and design described in section 1.3 is not precise enough to define on what categories of models feedback can be given. To describe more precisely on what categories of models feedback can be given, we use a categorization that distinguishes the following three dimensions [49] (see figure 1.3):

- *Reality* – Does the model represent a software system or an application domain? In the first case, we talk about a *system model*. A system can be an existing system, a part of it, or a system under development. In the second case, we talk about a *domain model*, i.e. that portion of reality that affects and is affected by the software system. Usually, the system model and domain model will have some overlap, i.e. both models will share some concepts and/or descriptions, while others exist only in one of the two models.

- *Purpose* – Is the model a specification of a domain/system to be built or a description of an existing domain/system? In the first case, a *specification*, the model is used as a form of forward engineering, i.e. specifies something that must exist, and can be used as a template to guide the construction of the system. Furthermore, it is possible to reason about the system/process before actually constructing it. In this case, we mainly talk about a *design model*. In the second case, a *description*, the model is used in a way of reverse engineering, i.e. the model is a conceptual tool to understand an existing system/process that has to be, for example, maintained or improved. In this case, we mainly talk about an *analysis model*.

- *Abstraction* – Is the model an abstract representation (black box model or logical view) or a concrete representation (white box model or implementation view)? In the first case, an *abstract representation*, we talk about a *logical model* which describes the system/process in terms of *what* it must do (requirements of the system). In the second case, a *concrete representation*, we talk about an *implementation model* which describes how the requirements are met by the implementation.

  It should be noticed that the degree of abstraction is relative, i.e. a model is an abstraction with respect to some other model [79]. For example, a UML specification of a business system is an abstraction of the implementation of that system, but the UML specification itself is a concrete realization of the use cases describing the functionality

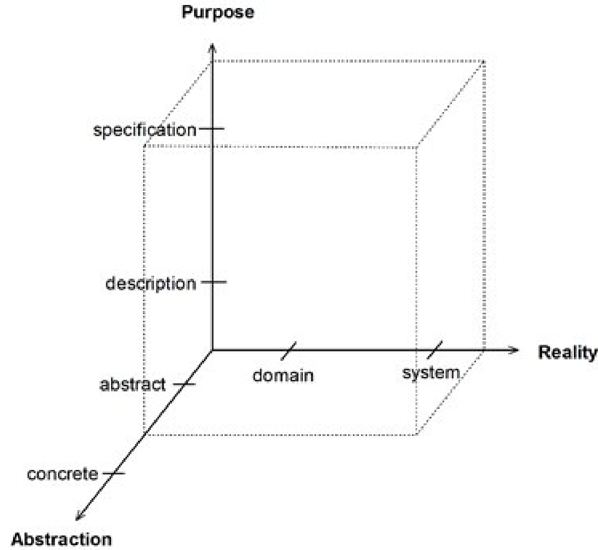


Figure 1.3: Three orthogonal dimensions (source: Journal of Object Technology, 2009, volume 8, No 1, page 109)

offered by the system. Furthermore, the level of abstraction is gradual. For example, by adding extra classes not part of the problem domain, a (abstract) domain model evolves into a (less abstract and more concrete) design model.

The three dimensions give rise to eight categories of models. In a typical (greenfield) software development project, only three of these categories are relevant.

1. First, an *abstract domain description* is developed to understand and describe the application domain, eventually completed with an abstract description of the business processes involved. Examples are a UML domain class diagram [85] and a Petri net as business process description [1].

2. Second, an *abstract system specification* is developed to specify what the software system should do. Examples are a UML use case model, a UML system sequence diagram, and a UML class diagram as basis for a design model, thus without taking new artefacts introduced for technical reasons as performance and platform choices [85].

3. Third, a *concrete system specification* is developed to specify how the software system should implement what it should do taking into account, for example, performance issues and the characteristics of the platform chosen. An example is a UML design model [85]. After the concrete specification is finished, the specification can be implemented using a programming language.

It is the third category where it is often difficult to give valuable feedback, i.e. no standard models or strategies exist and design choices often depend on the actual situation and are affected by aspects as performance, costs, maintainability, platform choice, etcetera. The cube-shape in figure 1.3, depicted by the dotted lines, defines the categories of models we will permit, i.e. all types of abstract models.

Remark. In the 'greenfield' example above, the UML class diagram notation is used to express three types of models, namely an abstract domain model (first step), an abstract system specification (second step), and a concrete system specification (third step). Each of these models serves a particular *perspective* and plays a certain *role* in the analysis and design process [130]. For example, in the first step the model involves the perspective of people working in a particular business process and has the role of describing that part of reality, whereas in the third step the model involves the perspective of code implementation and has the role of specifying the system's functions and architecture. By a number of endogenous vertical model transformations, the abstract domain model describing the application domain is transformed into a concrete system specification. These model transformations entail a path through the three dimensional space in figure 1.3. In a final step, the concrete system specification can be transformed into code skeletons which can be further implemented.

1.6 Examples extended

The previous sections 1.3 and 1.4 described what modeling and feedback mean. We now extend our examples, i.e. classify both models and the languages used, show the strategy or schema used to develop the model, and describe the type of feedback we want to give.

Example one: Modeling an XML content model. An XML content model is an example of a structural model, specifying the possible sequences of child elements of an element, and can be used as analysis model as well

as a design model. We distinguish two situations:

Situation 1: As stated in section 1.2, the course XML: Theory and applications does not describe a method for modeling XML content models. Assuming this situation, where the student tries to solve the problem at home using pen and paper, we are not able to give feedback at all: not at the level of the 'steps' followed, not about the efficiency of the method used, nor about the correctness of the resulting model. The student is left with all his or her questions.

Situation 2: As is described in chapter 6, we distinguish between *precise* and *correct* XML content models. Precise content models describe exactly the set of allowed sequences of XML elements, but nothing more, whereas correct models describe at least the sequences of XML elements we want to have. For both types of models, we have described a strategy for developing such models. Here, we focus on precise models only.

The method for modeling precise models consists of a set of rewrite rules, a strategy specifying in which order the rules should be applied, and a goal to reach. The method uses the regular expression language [66], which is a mathematical language with a clear syntax and semantics. For modeling a precise XML content model, only the following three rewrite rules are needed[4]:

$$R \mid S = S \mid R \quad (1a)$$
$$R? = \epsilon \mid R \quad (1b)$$
$$RS \mid RT = R(S \mid T) \quad (1c)$$

The goal to reach is a deterministic model. A content model is deterministic if an XML processor can check an XML document against a DTD without looking forward in the document (i.e. inspecting only the current element). For example, the content model $(a, b) \mid (a, c)$ is not deterministic. After applying rule 1c the model equals $a, (b \mid c)$ and this model is determinsitic. A simplified version of the strategy for reaching a deterministic precise model is:

Step 1. Write down the element content as a number of choices based on the example XML file.
Step 2. Remove non-determinism by applying rule 1c and, if necessary, rule 1a until the model is deterministic.
Step 3. Remove all occurrences of ϵ, the symbol which represents empty

[4] Here we present only the main steps. For more details, see chapter 6.

content, by applying rule 1b until all ϵ-symbols are removed.

Notice that in the first step an initial model is made, whereas the steps two and three are a matter of model transformations.

Now, the answer could show a correct model as well as a derivation to reach this model. The following example shows a correct derivation following the strategy. The first expression is the element content as a number of choices. The standard precedence levels apply: the unary operator ? binds stronger than sequence, which binds stronger than choice.

$$\begin{array}{rcll} a,b \mid a,b,a,b \mid a,b,c \mid \epsilon & = & a,b,(\epsilon \mid a,b \mid c) \mid \epsilon & \text{(1c)} \\ & = & a,b,(a,b \mid c)? \mid \epsilon & \text{(1b)} \\ & = & (a,b,(a,b \mid c)?)? & \text{(1b)} \end{array}$$

Suppose the student's derivation was:

$$\begin{array}{rcll} a,b \mid a,b,a,b \mid a,b,c \mid \epsilon & = & a,(b \mid b,a,b \mid b,c) \mid \epsilon & \text{(1c)} \\ & = & a,(b,(\epsilon \mid a,b \mid c) \mid \epsilon\text{''})\text{''} & \text{(1c)} \\ & = & a,(b,(a,b \mid c)? \mid \epsilon) & \text{(1b)} \\ & = & a,(b,(a,b \mid c)?)? & \text{(1b)} \end{array}$$

Notice that the student has made a mistake. In the second step, rule 1c is erroneously applied (the last bracket, marked by double quotes, is placed erroneously). Due to this erroneous application the third, fourth and final models are not correct.

The student can now see in addition to a correct model the derivation reaching this correct model. But again, in the situation where the student solves the problem at home using pen and paper he or she cannot detect automatically the mistake made; for the student it might be just another derivation. Moreover, the question whether the student's model is correct is not answered. The student may determine that the start of his or her derivation can be more efficient by factor out sub-expression a, b instead of a only.

The conclusion must be that the situation is less improved. Only in the situation in which the student starts the derivation in the same way, he/she is helped. In all other cases, the student misses information. One solution could be to show all possible derivations. However this is not always possible and is at least very labor intensive.

As a final remark, in a situation where multiple sequences of a, b are expected, the introduction of sub-expression $(a, b)^*$ is a consideration. The final model then will be something like $(a, b)^*, c$, which is an example of a correct model. Notice that the introduction of the star-operator is not a logical step but a design decision.

Example two: Modeling a domain diagram. The domain model using the UML class diagram notation is a structural model, i.e. specifies which classes exist in the domain of interest, the attributes of each class, and the associations between these classes. A domain model can be used as analysis model (describing the domain as-is) as well as a design model (specifying the domain as to-be).

Developing a domain model is not an algorithmic activity. There is no set of rules and corresponding strategy to develop a domain model with a certain result. As a result, we are not able to give feedback on the model development steps. Instead, some rules of thumb can be used guiding the developer in a certain direction. Furthermore, by model transformations as refactoring and refinement, the model can be improved and concretesized. As an example, a general schema for creating a domain model is [85]:

1. find the conceptual classes;
2. draw them as classes in a UML class diagram;
3. add associations and attributes.

Additional guidelines exist, for example the Noun phrase analysis or Linguistic analysis [2, 85], the Commonality and variability analysis [135] for finding conceptual classes, and patterns as the parent-child pattern, collection pattern, and sample pattern [81]. As we will see in chapter 2 and chapter 5 it is possible to give feedback on syntactic errors, meta-model errors (for example inconsistencies and redundancy in the model), and the semantics of the model (Does the model accurately reflect the part of reality it refers to?). However, in a pen-and-paper situation at home, the student is left with the question: 'Is my diagram correct too?'

1.7 Summary

We have described what modeling means and discussed some categories of models. Modeling is by nature constructive and a stepwise activity. For some type of models, there exists a set of rules and a strategy that describes the order in which the rules must be applied in order to reach a correct model. For other types of models, there exits only some rules of thumb. In the first case we are able to give feedback on the level of steps as well as on the correctness of the final model. In the second case, only feedback on the level of the correctness of the final model can be given. For feedback on

the final model, we need a standard model to compare with the student's model.

Important is the distinction between analysis and design models. As long as a model is abstract, we are able to give feedback. In cases of concrete models, design choices are made which often depend on the actual situation.

We have shown on the basis of two examples that, when a student is developing a model, immediate feedback is needed to answer questions a student has and to correct mistakes made by the student. In a classroom situation, it is the lecturer who provides this immediate feedback. In the current situation of distance education, immediate feedback is missed. For each exercise, only one or a few example answers are available which do not give answer on all questions a student could have.

What is needed is a system in which a student can develop a model and then receive automatically generated feedback on the level of the steps according to a strategy (if possible) and/or the resulting model.

1.8 This thesis

This thesis introduces a framework for automatic feedback generation during modeling activities. In this thesis, we do not have the intention to develop a complete system. Instead, subsequent chapters discuss some aspects.

Chapter 2 presents the functions and types of knowledge we need to generating feedback for modeling activities. We will distinguish between well-defined and ill-defined domains and tasks. After that, the framework and its components are described. The framework uses a number of ontologies for specifying rules, strategies, rules of thumb and standard models to compare with the student's final model. The framework distinguishes a student environment and an author environment. In the environment for students, students do modeling and receive feedback. In the environment for authors, authors specify exercises, feedback, and other learning materials. After the framework is introduced, related work is discussed and the research questions are formulated which will be treated in subsequent chapters.

Chapter 3 takes one type of models, namely solving linear equations. Solving linear equations is an example of a well-defined domain, i.e. there is a set of rewrite rules to rewrite terms into other terms, a strategy and a well-defined goal. We show how we can give feedback about syntactic errors, about several kinds of semantic errors, and about progression towards a solution. The framework explicitly uses the structure in the data to produce feedback.

Chapter 4 focuses on an aspect in the author environment. Course material for electronic learning environments is often structured using ontology and schema languages. During the specification and development of this material, many mistakes and errors can be made. In this chapter, schema analysis as a technique to analyze structured documents is introduced, and to point out a number of possible mistakes introduced by an author during authoring. With this technique, we are able to produce valuable feedback. We show the technique at work using six categories of mistakes and two types of schemata.

In chapter 5 we study a domain that is less structured, namely that of developing data models. Often, students have to learn several data model languages, for example a formal as well as an informal one. In this chapter, mechanisms are described for producing feedback about syntactic as well as semantic errors. Furthermore, the framework supports several model languages.

Being able to produce feedback requires a standard model to compare with, or preferably, a well-defined set of rules and a strategy describing in which order the rules must be applied. In many educational settings, this is not the situation. Some domains are inherently as unstructured as, for example, modeling a domain model is. In other domains that are more or less structured, the rules, strategy or rules of thumb are not described. Modeling XML content models was an example of the last case. Recently, we developed a general approach for modeling XML content models. To show what this means, chapter 6 describes this approach. Thinking about the goal to reach when modeling an XML content model, it turns out that there are two types of content models. For one type, precise models, we can define a set of rewrite rules and a strategy. For the other type, correct models, we can define a set of rewrite rules, but we cannot define a strategy. Instead, parts of the strategy are in the form of rules of thumb.

1.9 Origin of chapters

This thesis is largely based on a number of reviewed and published publications. Some parts of the papers have been revised and rewritten. The chapters of this thesis are based on the following publications:

Chapter 2. Passier, H. and Jeuring, J. (2004). Ontology based feedback generation in design-oriented e-Learning systems. In P. Isaias, P. Kommers and M. McPherson (editors), Proceedings of the IADIS International conference, e- Society, volume II, pages 992–996 [123].

The candidate is the main author of this paper.

Chapter 3. Passier, H. and Jeuring, J. (2006). Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, O. Caprotti, editors, Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT, pages 53–68, Oy WebALT Inc. [126].
The candidate has implemented the solver and analyzer of the feedback engine. The user interface has been implemented by the co-author. The paper is written in collaboration with the co-author.

Chapter 4. Passier, H. and Jeuring, J. (2005). Using Schema Analysis for Feedback in Authoring Tools for Learning Environments. Short version in Proceedings of the 12th International Conference on Artificial Intelligence in Education, AIED 2005 [124]. Extended version in A. Cristea, R. Carro, and F. Garzotto, editors, Proceedings of the Third International Workshop on Authoring of Adaptive and Adaptable Educational Hypermedia, A3EH 2005, pages 13–20 [125].
The candidate implemented Haskell functions himself and is the main author of the paper.

Chapter 5. Passier, H. (2008). A framework for feedback in e-learning systems for data modeling. In Proceedings of the IADIS International Conference, e-Society [119].
The candidate implemented Haskell functions himself and is the author of the paper.

Chapter 6. Passier, H. and Heeren. B. (2011). Modeling XML content models explained. In P. Kommers and N. Bessis, and P. Isaias, editors, Proceedings of the Internet Applications and Research 2011 [122].
The candidate started this research resulting in Notes on Modeling XML Element Content Models [120]. The final paper is written in collaboration with the co-author and available in an extended version [121].

Chapter 2

A framework for practicing modeling

2.1 Introduction

This chapter introduces a framework for an intelligent tutoring system in which a student can *practice modeling* and receives *immediate feedback*. The feedback is automatically generated, concerns the syntax and semantics of the model, and, if a strategy exists, is about the correctness of the steps according to this strategy. The purpose of this framework is to contextualize the research questions. We do not have the intention to fully implement this framework as part of this thesis; only some aspects are implemented.

An intelligent tutoring system is a type of e-learning system. Today, e-learning is widely applied in situations of distance eduation [56]. There exist many definitions of what e-learning means [56, 110]. E-learning is understood to mean all forms of internet enabled and/or computer supported learning. The term refers to the use of computer and internet technologies to create, deliver, manage and/or support learning content as well as the learning processes including, for example, practicing. E-learning can involve complete courses where almost all aspects of learning take place. On the other end of the spectrum, the learning process can take place in a traditional face to face class room situation where, for example, only practicing of what has been learned is supported by an e-learning system.

In e-learning systems *almost* all aspects of learning can take place. One

---

This chapter is based on: Passier, H. and Jeuring, J. (2004). Ontology based feedback generation in design-oriented e-Learning systems. In P. Isaias, P. Kommers and M. McPherson (editors), Proceedings of the IADIS International conference, e- Society, volume II, pages 992–996. [123]

exception is immediate feedback. As E.H. Mory [113] observed, there is a frequent lack of feedback in electronic learning environment courses in higher education and almost all feedback is related to question-answer situations and is hard coded. Exceptions are the environments based on social constructivism and serious gaming. In environments based on social constructivism, learners solve complex problems through social negotiations between equal (human) peers in a contextual setting [42]. Feedback occurs in the form of discussions among learners and through comparisons of internally structured knowledge [113]. In serious games, such as a flight simulator for training pilots, feedback consists of what a pilot experiences and is seamlessly integrated in the game context [29].

Intelligent tutoring systems are computer systems that aim to provide immediate and customized instruction or feedback to learners [37]. In the rest of this thesis, the term intelligent tutoring system (ITS) is used. During the last two decades, many ITSs are developed in well-defined domains such as mathematics and programming [88]. ITSs are almost all based on the assumption that students have learned the declarative knowledge and procedural knowledge from direct instruction, such as books and lectures. After that, the ITS is used for *practicing skills* [80]. As far as we know, ITSs for modeling are scarce. In section 2.5 we will give an overview of relevant ITSs we know about.

Besides the ITS for practicing modeling, the framework consists of an *author environment* in which a lecturer can specify modeling exercises. Designing and specifying course material have much in common with modeling [143]. For example, when an author develops his or her course, he/she has to choose, develop and/or adapt task ontologies and domain ontologies and related material like examples and definitions [106, 129]. Ontologies are types of models and as such make use of some notations with a certain syntax and semantics.

To define the framework, we have to declare in general terms the functional components and the types of knowledge that play a role in determining the information a student can use to improve or optimize his or her modeling processes. As Bundy described in his book *The computer modeling of mathematical reasoning* [27]: 'Whatever aspect of intelligence you attempt to model in a computer program the same needs arise over and over again:

- The need to have knowledge about the domain.
- The need to reason with that knowledge.
- The need for knowledge about how to direct or guide that reasoning.'

We explore from different perspectives the knowledge types that play a role in the context of modeling, namely from the definition of feedback, control theory, problem solving, modeling, and education. After that, we describe the framework, list related work, and list the research questions.

The example used. The different types of knowledge are introduced on the basis of a simple example, namely solving a system of linear equations. This is an example of secondary school mathematics. An example of a system of linear equations is:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - y \end{cases}$$

A student has to rewrite these equations into a form with a variable to the left of the equal symbols and a constant to the right of the equal symbols, in this case $x = 2$ and $y = 1$. We call this the solution, or semantics, of the system. Notice that this solution remains unchanged during rewriting, otherwise a rewrite mistake has happened.

The solving process could be, using the *substitution method* as strategy, as follows:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - y \end{cases} \Leftrightarrow$$

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - (x - 1) \end{cases} \Leftrightarrow$$

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - x + 1 \end{cases} \Leftrightarrow$$

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 3 - x \end{cases} \Leftrightarrow$$

$$\begin{cases} y & = x - 1 \\ 1\frac{1}{2} \cdot x & = 3 \end{cases} \Leftrightarrow$$

$$\begin{cases} y &= x - 1 \\ x &= 2 \end{cases} \Leftrightarrow$$

$$\begin{cases} y &= 2 - 1 \\ x &= 2 \end{cases} \Leftrightarrow$$

$$\begin{cases} y &= 1 \\ x &= 2 \end{cases}$$

Solving a system of linear equations is a stepwise activity. In each step, the student applies one or more rewrite rules. We call a rule application a rewrite step. An example of a rule is the substitution rule, which is applied in the first step: variable y in the second equation is substituted by the value of variable y in the first equation, namely $x - 1$. Another example of a rule is the addition of two constants resulting in one new constant. This rule is applied in the third step, where the constants 2 and 1 are added.

The substitution method is an example of a strategy. A *strategy* describes which sequences of rule applications are allowed. The substitution method prescribes to rewrite one equation in a form of $x = \ldots y \ldots$ or $y = \ldots x \ldots$ and to substitute, by applying the substitution rule, the right hand side of one of them in the other equation. In this case of solving a system of linear equations, if there is a solution, the strategy guarantees the solution. Solving a system of linear equations is discussed in more detail in the next chapter.

Rewriting a mathematical expression is a type of model transformation. Through rewriting, we *transform* an expression into another form which exhibits certain properties. In case of rewriting a system of linear equations, we want to express the solution of the system. Rewriting a mathematical expression assumes a start model, in our case a set of linear equations. These equations could represent an operational problem, as for example a stock problem. Using the model, optimal delivery periods and order sizes could be determined.

2.2 Main concepts and functions

In the previous chapter, we have seen that feedback is information presented in such a way that it allows comparison between an actual outcome (or performance) and a desired outcome (or some set standard of performance). From this definition we can identify two main concepts, namely:

- an actual outcome;

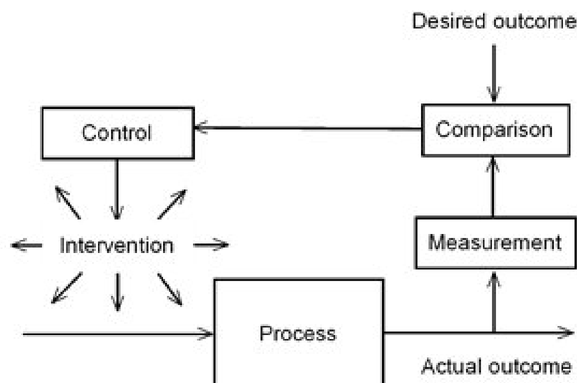


Figure 2.1: Symbolic representation of feedback

- a desired outcome;

From control theory, we know that we need three functions to generate feedback, namely:

- a measurement function;
- a comparison function;
- a control function.

Figure 2.1 shows these concepts and functions and the relations between them. The goal is to control a (central) process, measured by the actual outcome. The actual outcome is compared to a desired outcome. If there is a deviation with respect to the desired outcome, the unit of control performs some intervention at the input side of the process or in the process itself [153].

In the next paragraphs, we will briefly describe these concepts and functions.

Actual outcome. Each rewrite step results in a new system of equations. We call each of these systems an actual outcome. For example, after the student has finished the first rewrite step in our example, the system of equations is:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - (x - 1) \end{cases}$$

As can be seen, the student has not made a mistake, i.e. the sub-expression $2 - y$ is correctly rewritten into $2 - (x - 1)$.

Solving a system of linear equations is a stepwise activity according to a strategy (in this case the substitution method). As a result, the solving process consists of a *sequence* of systems of equations, where each system is compared to a desired outcome. This is not always the case. For example, for modeling a UML class diagram there is no strategy, only some rules of thumb. As a result, a student often submits only a final UML class diagram instead of a sequence of diagrams.

Measurement function. The measurement unit prepares an actual outcome for comparison with the desired outcome. For example, the measurement unit parses the system of equations for internal processing. The measurement function takes always *one* model for comparison to the desired outcome. In case of an ITS, measurement takes place when a student submits a model for evaluation.

The desired outcome. The desired outcome for a system of linear equations is described by three parts: (1) the system of equations is *syntactically* correct, (2) the *semantics* of the actual system of equations equal the semantics of the previous system of equations, and (3) the actual system of equations can be obtained by applying a rewrite rule according to a *strategy* for solving a system of linear equations.

If the strategy consists of one sequence of steps, there is only one desired outcome for each actual outcome. This is, however, seldom the case. Looking at the first system of equations, applying the substitution rule was obvious but other rewritings are certainly possible. For example, the student could multiply both sides of the second equation by 2, resulting in the system:

$$\begin{cases} y & = x - 1 \\ x & = 4 - 2 \cdot y \end{cases}$$

Both systems, i.e. both rewritings, are consistent with the substitution method. As a result, in case of solving a system of linear equations, the desired outcome often consists of a number of systems of equations, namely all systems that can be obtained from the previous system by correctly applying a rewrite step according to the strategy. An exception is the final system, i.e. the solution, which is unique. As a result, the desired outcome for the final system consists of one member.

Generally, exactly one desired outcome is uncommon in modeling. Often, several outcomes are possible as is the case in modeling a UML class

diagram. Furthermore, for most model types there does not exist a fine grained strategy. Often, only some rules of thumb are available.

The comparison function. The comparison unit compares the actual outcome with the desired outcome. In case of solving a system of linear equations, the output of the comparison unit could be one out of five options:

- If the actual system of equations is syntactically incorrect, then the output of the comparison unit is a syntax error code.
- If the actual system of equations is syntactically correct and the semantics do not equal the semantics of the previous system, then the comparison unit *analyses* this situation and tries to detect which rule the student has probably applied.

  Suppose, the student has rewritten the first system of equations into the following one:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - x - 1 \end{cases}$$

  The student has tried to apply the substitution rule, but has forgotten to place the sub-expression $x - 1$ inside brackets.

  The output is a semantic error code, including information about the erroneously applied rule.

- If the actual system of equations is syntactically correct, the semantics equal the semantics of the previous system, and the system is in the final form, i.e. $x = c_1$, $y = c_2$, where c_1 and c_2 are constants, then the solving process is finished. The output of the comparison unit is a ready-code.
- If the actual system of equations is syntactically correct, the semantics equal the semantics of the previous system, the system is not in the final form, and the actual system can be obtained by applying a rewrite rule according to the strategy allowed, the output of the comparison unit is a code representing that the step was correct, but the solving process is not yet finished.
- If the actual system of equations is syntactically correct, the semantics equal the semantics of the previous system, the system is not in the final form, and the actual system can not be obtained by applying a rewrite rule according to the strategy allowed, the comparison unit produces an error code indicating the deviation with respect to the strategy.

The control function. Depending on the result of the comparison unit, the control unit computes the content of the feedback message:

- In the first case the message could be a report about the syntax error, for example: 'You miss a bracket.'
- In the second case the message could report about the rule which is probably erroneously applied. Continuing our example: 'You have made a mistake: Since variable y has disappeared from the equation $\frac{1}{2} \cdot x = 2 - y$ we assume you have tried to apply the substitution rule. Correctly applying the substitution rule for y results in $\frac{1}{2} \cdot x = 2 - (x - 1)$. Is this what you meant?'
- In the third case, the message could be something like: 'You have solved the system. Congratulations!'
- In the fourth case, the message could say: 'That was a correct rewrite step. Proceed further.'
- In the last case, the message could be: 'In itself, it was a correct step, but you deviate from the strategy.'

Notice that the comparison as well as the control unit need knowledge about the syntax of the expression language, the previous expression, the actual expression, and the strategy.

From *problem solving theory* we know a problem consists of four components, namely an initial state, a goal state, operators or rules that can be used to reach the goal state, and a task environment that the solver is working in [43, 143].

The initial state. The initial state contains the problem description and the constraints that must be satisfied. In our example, the initial state consists of the exercise text: Solve the following system of linear equations:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - y \end{cases}$$

An example of a constraint is that the substitution method must be used.

In our example, the initial state is well-defined, i.e. is not ambiguous. There are many examples where this is not the case. In modeling tasks the problem description in natural language is often ambiguous. As a known example, the domain description 'The hunter shoots the rabbit with his gun' could be interpreted in three ways.

The goal state The goal state is the state in which the problem is solved. Sometimes the goal state is well-defined, as for example in solving a system of linear equations: all equations are in a form with a variable to the left of the equal symbols and a constant to the right of the equal symbols. Again, this is not always the case. In for example modeling a UML class diagram, many class diagrams may form a valid solution and there is often not one best solution. In such cases, we can describe a set of properties (or constraints) that must hold. These properties accept a number of final solutions.

Operators or rules. Generally, operators and rules are descriptions of actions in terms of which state will be reached by carrying out the action in a particular state.

An example of a rule is the substitution rule. This rule can be applied, in case of two equations, if one equation has the form $x = \dots y \dots$ and the other equation has both variables x and y. Substituting the right hand side of the first equation into the second equation decreases the number of variables in the second equation.

The substitution rule is an example of a very precise rule with clear semantics, i.e. if the prerequisites are satisfied, the action results in a desired state. Again, this is often not the case in modeling. Often, there are only some rules of thumb, as for example 'list all potential class-names' for modeling a UML class diagram. Applying such a rule will result in many potential outcomes.

Task environment. The task environment consists of the features that can either directly or indirectly constrain or suggest different ways of solving a problem. In our case, the environment could be at home, in a class situation with or without a tutor, in an ITS, and with or without supporting material as a text book.

Problem and modeling types. Which types of problems or modeling tasks can be distinguished? Problems are described as varying on a continuum from well-defined to ill-defined, or well-structured to ill-structured [64, 76, 152]. It should be noticed that the terms 'defined' and 'structured' are used interchangeably in the literature [88].

Mitrovic and Weeransinghe [104] point out, in the context of modeling activities in an ITS, the importance of distinguishing between *definedness of*

the modeling language[1][2] and *definedness of the problem solving task*. With *problem solving task* they refer to the procedure or strategy of how to solve the problem, including a description of the initial state, the goal state, and how to evaluate the solution for correctness. They propose two orthogonal dimensions, one for the degree of definedness of the model language and one for the degree of definedness of the problem solving task. The two dimensions are continuous, i.e. the spectra arranging languages and tasks from ill- to well-defined ones. These two dimensions result into three classes of problems (the fourth 'ill-defined modeling language - well-defined task' cannot occur):

- well-defined modeling language - well-defined problem solving task (for example solving a system of linear equations),
- well-defined modeling language - ill-defined problem solving task (for example conceptual database design expressed in ER-notation),
- ill-defined modeling language - ill-defined problem solving task (for example writing an essay), and

In our opinion, the distinction between modeling language and problem solving task (strategy, initial state, goal state, and evaluation procedure) is not distinctive enough. There are examples of modeling tasks for which there is a well-defined strategy, but the initial state can be ill-defined or well-defined. If the initial state is well-defined, the strategy can be used. If the initial state is ill-defined, the strategy can not be used without some preparative work.

For example, consider modeling a precise XML content model. As we have seen in the previous chapter, there is a well-defined strategy of how to model such an XML content model. This strategy needs a well-defined initial state: an example XML document. If we model a content model on the basis of an example XML document, the initial state is well-defined and the strategy can be used. On the other hand, if we have to model the XML content model on the basis of an ambiguous description in natural language, the initial state is ill-defined. Some preliminary work must be done, i.e. for every element we have to choose which sequences of child elements are

[1] Mitrovic and Weeransinghe use the term *problem domain* for *model language*. Because the term problem domain has another meaning in for example OO-modeling, we prefer to use the term model language.

[2] It should be noticed that the terms *language* and *domain* are used interchangeably. In the context of modeling we will use the term language. In the context of mathematics, we will use the term domain.

allowed. This preliminary work corresponds to 'developing a first model', as stated in the previous chapter, and is less defined, due to the ill-defined domain of interest and the choices which have to made.

To have the possibility to express this distinction, we add an extra dimension to the model, namely the *definedness of the domain of interest*. The degree of definedness of the domain of interest, as is presented to the problem solver, determines to a large extent the definedness of the initial state of the problem to solve. In summary, we distinguish the following three dimensions:

- *Modeling language* – The modeling language used.
- *Domain of interest* – A description of the domain of interest, including the initial state of the problem so solve, as is presented to the problem solver.
- *Problem solving task* – The strategy that should be used to solve the problem, a description of the initial state as precondition to use the strategy, the goal state to reach, and a procedure of how to evaluate the solution for correctness.

Figure 2.2 shows these three dimensions, where we have replaced Domain into Model language.

Remark. We make a distinction between the *real domain of interest* and the *domain of interest as is presented to a student*. By doing some preparatory work, an ill-defined domain of interest is transformed into a more well-defined domain. This preparatory work is in fact a movement along the axis Domain of interest from ill-defined to well-defined and is often an important first step in many it-projects. This movement reflects the first activities as domain analysis and requirements analysis. In an exercise environment, as modeling in an ITS is, by presenting a description of the domain of interest in a more *stylized* or *formalized* way, a modeling problem becomes less ill-defined and more well-defined. This preparatory work is done by the lecturer. Additional advantage is that by transforming an ill-defined domain into a well-defined domain, we are able to give more relevant feedback about the semantics of the model.

Remark. For some model transformations, namely if the semantics of the model does not change, the axis domain of interest does not play a role. For example, solving a system of equations assumes a start model, i.e. the

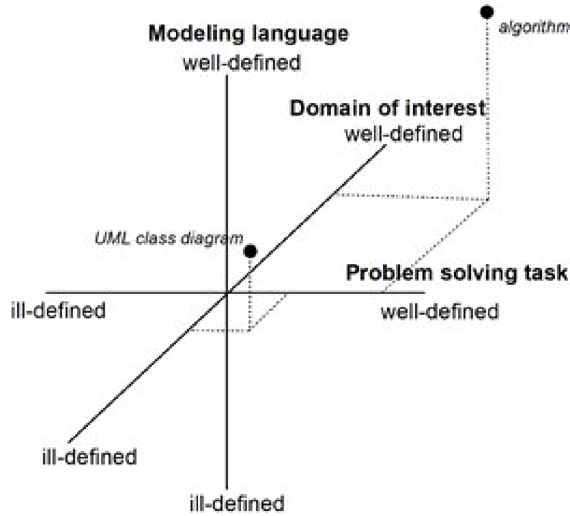


Figure 2.2: Problem types

first set of equations. This start model could represent a certain domain of interest, for example a particular logistic process, but could be a pure mathematical exercise in its own too. 'Solving the set of equations' is only determined by the definedness of the model language (the mathematical language of systems of equations) and the problem solving task (for example the substitution method).

Remark. According to Jonassen [76], these three dimensions are relative to the modeler or student, i.e. the familiarity with the task to solve, the familiarity with the modeling language to use, and the familiarity with the problem domain of interest.

We give some examples of relevant combinations:

- Well-defined modeling language, well-defined domain of interest, and well-defined task – An example is modeling a precise XML content model using the DTD content model language, based on one or more example XML documents. The task is an algorithm, as we have seen in the previous chapter. An algorithm is included in figure 2.2.

- Well-defined modeling language, well-defined domain of interest, and ill-defined task – An example is modeling a UML class diagram on the basis of a stylized textual description. Here we consider the UML-class

diagram notation as reasonably well-defined, i.e. the class diagram language is positioned on the well-defined half of the axis. The UML-language (UML 2), however, is not totally well-defined, but in some way imprecise and ambiguous [23].

- Well-defined modeling language, ill-defined domain of interest, and ill-defined task – An example is conceptual database modeling based on an ambiguous text using the the ER-notation.

The problem solving process, schemata and strategies. In a situation of a well-defined domain of interest, a well-defined language, and a well-defined task there often exists a well-defined strategy. Depending on the student's level of knowledge and skills, the student recognizes the situation, chooses the right strategy and performs this strategy. In a situation of an ill-defined domain of interest, a well-defined language, and an ill-defined task, there is no well-defined strategy. In that case, the student has to use a heuristic to develop the model asked.

Gick described a simplified problem solving process consisting of two sub-processes [52]:

1. generation of a problem representation or solution space (the problem solvers view of the problem);
2. a solutions process that involves a search through the solution space.

During the construction of the problem representation, certain features of the problem may activate knowledge in memory. The problem representation contains the start state including the obstacles, a description of the goal state including criteria for when the goal is reached, and a description of prior declarative knowledge, rules, and strategies that will aid in solution. A schema for that type of problem may then be activated.

A *schema* is defined as a cluster of knowledge related to a problem type. It contains information about the typical problem goal, constraints, and the strategy or strategies to solve the problem [52]. The *strategy*, or solution *procedure*, is a generalizable series of steps initiated in response to a particular class of circumstances to reach a specific goal. Strategies are sometimes strictly defined, i.e. all steps are included with no ambiguity in each step. Such a strictly defined strategy is called an *algorithm*. Strategies can be simple (a linear sequence of steps), or very complex (with many decision points, at which a solver must determine which of two or more situations exist and based on this determination which branch of the strategy will be followed) [137].

Strategies can be often combined. One situation is where two or more strategies exists for one and the same problem. An example is solving a system of linear equations for which four strategies exist [114]. The substitution and the combination method are strategies for solving a system of linear equations and can be used in combination, i.e. one or more steps from both strategies are applied alternately. Another situation is were strategies are combined in a way more complex problems can be solved. For example, the quadratic formula can be used for solving a quadratic equation. Factorization is a strategy for finding roots of a polynomial of degree greater than two. Both can be combined to solve a polynomial $P(x)$ of degree three: *first* search for a root r, *then* factorize $P(x)$ such that $p(x) = (x - r) \cdot P'(x)$, *followed* by applying the quadratic formula on $P'(x)$. Notice that producing feedback in these situations is extra complicated by the fact we have to detect which strategies are used and which part of which strategy is used when.

If schema activation occurs, the solver can apply the solution strategies contained in the schema. Schema driven strategies may be *general* or *domain specific*. An example of a general strategy is decomposition of the problem. An example of a domain specific strategy is modeling an XML content model.

If there is no appropriate schema activation, the problem solver proceeds to the second step and a search strategy is invoked. People often use heuristics for searching the problem space. A *heuristic* is a rule of thumb, also called a weak method, that will generally get one at the correct solution, does not guarantee the correct solution, and is independent of a particular problem [43, 96]. Examples of heuristics are: randomly pick up a reachable next state [43], Hill climbing [43], problem decomposition [43, 51], Mean-ends-analysis [43, 51], and reason by analogy [43, 128].

In order to produce feedback on modeling activities, an ITS must have different types of knowledge. Based on the sources described above, it is clear that the concepts schema, initial state, goal state, constraint, operator or rule, strategy, procedure and heuristic all play an important role in problem solving and modeling. The descriptions and relations between the concepts mentioned so far vary per author (see also section 2.4). In the rest of this thesis, we assume the following descriptions and relations:

- A *schema* is defined as a cluster of knowledge related to a problem type. It describes, if existing, the typical initial state, goal state, constraints, the operators or rules that may be applied, and the strategy to solve the problem

- A *strategy* describes how a problem should be solved. A strategy could

be domain specific or generally applicable. Furthermore, a strategy could be in the form of an algorithm, a procedure and a heuristic. Sometimes, a number of algorithms, procedures and heuristics exists which can be sometimes used in combination.

- A *procedure* is a description of steps or operator/rule applications that transforms an input into an output.
- An *algorithm* is a special form of a procedure, namely a description of steps or operator/rule applications that transforms a well-defined input (described by a pre-condition) into an well-defined output (described by a post condition). If the input satisfies the precondition, success of the transformation is guaranteed.
- A *heuristic* is a rule of thumb that guides a solver into a certain direction, but does not guarantee the correct solution.

Notice that describing the knowledge needed for modeling in such a way concurs with Anderson's first principle [4, 5]: if possible, the student's competence should be represented as a *production system*. The production system says which rules may be applied in which order. Only then an ITS is able to follow the student's solving process, can detect erroneous rule applications and missing rules, and can correct erroneous steps made. In cases the production system consists only of high level or heuristic rules the model can (only) test on these high level steps and the required qualities of the final model or sub-models.

Educational issues. The goal of learning problem solving is the ability to perform the solving schema as a whole. Often, these schemata are complex. Therefore, it is helpful to first practice the components apart of each other before moving to a complete schema. Some examples of practices for learning schemata are [137]:

- learning to determine if a strategy is required,
- learning to complete the steps in a strategy,
- learning to list the steps in a strategy, and
- learning to check the appropriateness of a completed strategy.

For learning solving problems, different *tutorial strategies* can be needed [54]. For example, for learning declarative knowledge a bottom-up approach, or

inductive learning, can be used as an educational strategy rather than a top-down approach or *deductive* learning. Specifically for solving complex problems, practice may *initially* include instructional guidance such as detailed hints, guiding questions, presentation of the rules, suggestions for strategies and information about the efficiency of the solution process. For example, after an erroneous rule application during solving a system of equations, the system could present the rule application in the context of another example. In later stages, if the student is more matured, hints could contain less detailed information. During initial stages of practice, feedback should be immediately available [137].

2.3 The framework

We imagine an environment for practicing modeling, in which:

- learners are able to develop models of certain domains using different types of languages;
- authors are able to develop modeling exercises;
- learners as well as authors receive semantically rich feedback during developing models and exercises based on different ontologies, i.e. a domain, a task, an educational and a feedback ontology.

The framework consists of four main components: a player for the learner, an authoring tool, a feedback engine and a set of ontologies as pluggable components. The player consists of a modeling environment in which a learner can develop models. The authoring tool consists of an authoring environment where the author develops modeling exercises and course related materials such as a domain ontology, a task ontology and feedback messages. The feedback engine automatically produces feedback to learners as well as to authors. The ontologies represent the modeling language, task, educational and feedback knowledge. Figure 2.3 gives an overview of the functional components of the framework.

The framework corresponds to the main components an ITS often consists of, namely a user interface (a layer in the student player and author environment), a domain model (corresponding with the task - and domain ontologies), a teaching model (corresponding with the educational and feedback ontologies), and a student model [116]. It should be noticed that the framework does not contain a student model. Tracing the students' progress over several modeling exercises does not have our attention in this thesis.

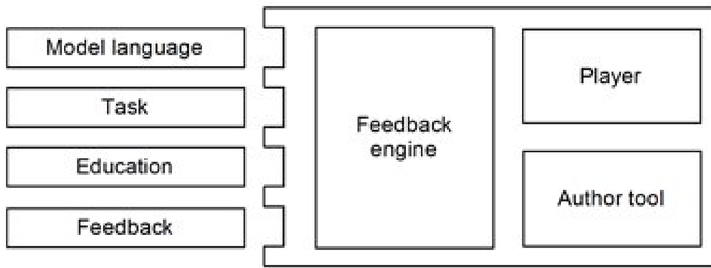


Figure 2.3: Functional architecture

Ontologies. The system contains several types of knowledge to produce semantically rich feedback. The knowledge is represented by ontologies. Generally, an ontology is: (1) a vocabulary, i.e. an informal description of the concepts, predicates, functions and constants of the domain of interest, and (2) a formal encoding of general knowledge about a domain in terms of axioms [131].

Declarative knowledge as conceptual and structural models can be represented by predicate calculus, conceptual graphs and even OO domain class notation, ER notation [141], and Resource Description Framework (RDF) [154]. Specifying causal models and strategies, i.e. algorithms, procedures and heuristics, can be represented by Petri nets, context free grammars and full programming languages [70, 141]. By describing the domain precisely, we are able to construct inference procedures for automatic derivation of consequences, i.e. in our case to produce feedback.

With ontologies as arguments, the different types of knowledge are considered as reusable components of knowledge. This is important, because the development of feedback mechanisms is time consuming and specialist work [106]. For example, a description of a domain of interest used in an exercise asking a student to make a UML class diagram, can be used is an exercise asking a student to make an ER-diagram too.

Based on the survey in the previous section, we distinguish four types of ontologies:

- *Modeling language* – The modeling language ontology describes the model language used. Examples are the DTD language for specifying XML content models and the UML class diagram language for specifying class diagrams.

- *Task* – The task ontology describes a schema to solve a particular problem. The schema describes in general terms how to approach the

problem, i.e., if existing, the initial state, constraints, operators or rules that may be applied, one or more strategies, and the goal state to reach. Each exercise could be considered as an instance of such a schema, i.e. each exercise describes the actual initial state, constraints, operators or rules and/or strategy that may be applied. If the goal state is well-defined, as for example is in solving a system of linear equations, the goal state could be described by one model solution. If the goal state is ill-defined, as for example is in modeling a UML domain diagram, the goal state could be described by a number of model solutions and/or a set of constraints that must be satisfied by the student's solution.

- *Education* – The education ontology describes educational issues such as the grain size of modeling steps allowed, to which extent the student should solve all the modeling steps himself, and which type of feedback is appropriate depending on maturity of the student.
- *Feedback* – The feedback ontology describes different types of feedback and feedback patterns during dialogs. Examples of immediate (corrective) feedback are a simple correct/incorrect or more complex feedback messages describing the correctness and efficiency of a step and to which extent the goal is reached. An example of delayed feedback is a report about the efficiency of the strategy the student has followed. Furthermore, the roles of feedback are described, as for example direct, inform, instructional, etcetera.

The author tool. In the author environment, an author develops and maintains modeling exercises. For each exercise, the author chooses a model language (for example the UML class diagram notation or the DTD content model language), describes the task in terms of a domain of interest and a problem description, chooses an appropriate schema consisting of for example a strategy of how the model should be developed, defines the goal state in terms of one or more models and/or a set of constrains, adjusts educational issues as for example the grain size of modeling steps allowed and the types of feedback which should be used. If the components needed already exist, they can be re-used. If necessary, these components can be adapted and new components can be defined.

The system performs many functions in the background that check on consistency, completeness, etcetera, of the various types of knowledge part of the exercises. These mechanisms allow the author to be flexible, for example, to enter what he/she wants in whatever order he/she wants.

The player. In the player a student practices modeling. The environment is designed as a complement to classroom teaching, i.e. it is assumed that students are already familiar with the fundamentals of the type of modeling to be practiced. Students work individually on a modeling exercise. The player is a problem solving environment in which a student constructs a model of a certain type that has to satisfy a given set of requirements. These requirements are presented in the form of a textual description.

If necessary, the system assists students during modeling and guides them towards the correct solution in the form of several types of feedback messages and hints. In cases of well-defined domains and tasks, the system can produce feedback after each step done by the student as well as after the final model is submitted by the student. In cases of less defined tasks, the system can produce feedback after the final model, or a part of the final model, is submitted. Furthermore, depending on the strategy, feedback about the steps according to the algorithm, procedure or heuristic can be produced.

The feedback engine. The feedback engine analyses the actual outcome of the student's model after the student submits his or her final model or partly finished model. If there exists for example an algorithm for solving the modeling problem, the engine can analyze the modeling process on a step-by-step basis.

The desired outcome is defined by the author. Depending on the definedness of the task, the desired outcome could be specified in terms of one or more final models, in terms of sequences of modeling steps, i.e. a strategy reaching a final model, and/or as a set of constraints the final model must satisfied.

The engine compares the actual outcome to the desired outcome in terms of syntax of the notation used, the semantics of the model, and, if possible, a strategy describing the modeling process in more or less detail. Each deviation is reported to a control function which formulates feedback messages. To construct feedback, the feedback engine uses the argument ontologies. Since the ontologies are arguments, the feedback engine does not have to be changed if an ontology is exchanged for another.

This framework, a general feedback engine and the use of ontologies as arguments, supports the constant requirement for flexibility, adaptability and reusability of knowledge structures in e-learning systems [116]. Authors and knowledge engineers can concentrate on ontology engineering and authoring of course material, while the general feedback engine remains the same.

2.4 Additional concepts

In this section, we explore in more depth the concepts used in the previous sections. This section provides some backgrounds and results mentioned in the literature about modeling and problem solving.

Problem solving. There exist several definitions of what a problem is and what problem solving means. One theory is based on the metaphor that how humans solve problems is like a computer program [43, 143]. According to this theory, a problem consists of four components, namely an initial state, a goal state, operators or rules that can be used to reach the goal state, and a task environment that the solver is working in. The task environment consists of the features that can either directly or indirectly constrain or suggest different ways of solving a problem. Problem solving then consists of defining the solution space after which solution paths are searched from the initial state to a goal state using the operators or rules [117, 143].

Problem types. Problems are described as varying on a continuum from well-defined (well-structured) to ill-defined (ill-structured). For example, Hicks [64] distinguishes between simple and complex problems (based on the structure of the problem), well-defined and ill-defined problems (based on whether problem solvers are confident about the direction of possible solutions), and tame and wicked problems (based on, for example, whether problems have a definitive problem description, the possibility of knowing the best solution is reached, whether possible solutions are not true or false, but somewhere between good and bad, they have a number of possible solutions, etcetera). It should be noticed that the terms well-defined, well-structured, ill-defined, and ill-structured are used interchangeably in the literature [88].

A well-defined domain is one in which there is a systematic way to determine when a proposed solution is acceptable; on the other hand ill-defined domains lack such a procedure [88].

To be more precise to which extent problems are solvable, Jonassen described a typology of problem solving [76]. Jonassen classifies problems along three dimensions: problem type, problem representation, and individual differences:

- The first dimension, the *problem type*, is a problem internal characteristic and varies by structure, complexity, and abstractness. The *structure* of a problem ranges from well-structured to ill-structured problems. Well-structured problems present, for example, all elements

of the problem, have a probable solution, engage the application of a limited number of rules, principles or operators that are organized in a predictive and prescriptive arrangement, and possess correct and convergent answers. On the other hand, ill-structured problems are in all of these characteristics the opposite. *Complexity* is especially concerned with how many components are in the problem, how these components interact, and how consistently they behave. *Abstractness* makes differences between domain specific (or concrete) and domain independent (or abstract) problems.

- The second dimensions, *problem representation*, is about how the problem is presented to the problem solver.
- The third dimension, *individual differences*, is about, for example, the familiarity of the problem solver with the problem type, the solvers level of domain knowledge, and domain specific thinking skills.

The range of problem solving types is described by a continuum from well-structured to ill-structured, abstract to concrete, and simple to complex problems: logical problem, algorithm, story problem, rule-using problem, decision making, trouble shooting, diagnoses/solution, tactical-strategic performance, situated case, design, and dilemmas. In the context of modeling within computer science, the types algorithm, logical problem, and design are of particular interest.

- *Algorithmic problems* can be solved using a finite and rigid set of procedures, with limited, predictive decisions. Examples are modeling a precise XML content model and solving a linear system of n equations with n variables. Solving these problems requires to select and apply the correct sequence of operators or rules to the formula. These type of problems are well-structured, often abstract, can be simple or very complex, and can be solved correctly.
- *Logical problems* tend to be abstract tests of logic that puzzle the student. In this type of problems, there is a specific method of reasoning that will yield the most efficient solution. Logical problems can be complex. Examples of complex problems are the games Bridge and chess, which employ more complex rules and constrains. Often, these more complex forms of problems also require other forms of problem solving, including diagnoses/solution, and perhaps design. An efficient solution is often possible. These type of problems are between well- and ill-structured, often abstract, and can vary from simple to very complex.

- *Design problems* are at the other end of the continuum. Usually, most design problems have multiple if not infinitely many solutions, the criteria for the best solution are not always obvious, there exist no ready-made procedures, and the constrains and start situation might be partly unknown. These type of problems are often ill-structured, domain dependent, and are often complex.

The typology of Jonassen corresponds with the taxonomy of problems defined by Van Gundy [152]. This taxonomy consists of three types classified on the extent to which they are structured:

- *Well-structured problems* – These problems are characterized by the availability of all information needed to close the problem gap. These type of problems can be often solved applying a schema or procedure. The algorithmic problems in Jonassen's typology are a type of well-structured problems.

- *Semi-structured problems* – These problems provide the solver with some information, but it is not enough or there are some unclear points. Uncertainty exists about the current state, the goal, and/or which procedure to use. Here heuristics can be used. The logical problems in Jonassen's typology are a type of semi-structured problems.

- *Ill-structured problems* – These problems provide the solver with little or no information on the best way of developing a solution. Because no clear procedure exists for how to solve the problem, the solver should improvise and develop a custom made solution. In this class of problems creativity plays an important role. The design problems in Jonassen's typology are a type of ill-structured problems.

In the survey so far, the degree of definedness (or structuredness) is particularly related to the problem domain. Mitrovic and Weeransinghe [104] point out the importance of distinguishing between definedness of the problem domain and the problem solving task. They propose two orthogonal dimensions, one of the degree of definedness of the problem domain and one of the definedness of the task. The two dimensions represent four classes of problem solving tasks:

- *Well-defined domain - well-defined task*: Instances of this class are most covered by ITSs. Examples of domains are areas of mathematics and physics, such as systems of linear equations and fractions. Tasks

in this domain are algorithmic in nature, for example solving a system of n linear equations with n unknowns, and adding and reducing fractions.

Another example is developing precise XML content models. The XML content model language, i.e. the regular expression language, is well-defined. The task is well-defined too, i.e. there exists an algorithm for constructing such a model.

- *Well-defined domain - ill-defined task*: Instances of this class are less covered by ITSs. An example is conceptual database design expressed in terms of the ER data model. The ER data model language is well-defined: it consists of a small number of components with a well-defined syntax and semantics. However, the task conceptual database modeling is ill-defined: the initial state is usually underspecified and ambiguous, there exists no algorithm, and the goal state is underspecified. Furthermore, the domain of interest modeled in an ER-diagram could be ill-structured or ill-defined too.

  Another example is developing a correct XML content model. The XML content model language is well-defined. Furthermore, the start state is well-defined. However, the task is ill-defined, i.e. there is no algorithm to transform the start state into a desired goal state. Instead, as we will see in chapter 6, there are rules of thumb for solving this problem.

- *Ill-defined domain - ill-defined task*: Instances of this class are rarely covered by ITSs. An examples is essay writing, for which there are only some global rules for how to structure an essay and how to present arguments.

- *Ill-defined domain - well-defined task*: There are no examples of this class.

To which class a problem solving task belongs, depends on the degree of definedness of the initial state and the goal state, and the availability of a problem solving procedure for that problem.

Lynch et al. [88] distinguish five characteristics, namely verifiability (to which extent are there valid arguments for or against a solution), open-textured concepts (to which extent are there abstract concepts that lack an absolute definition), overlapping problems (to which extent are sub-problems after decomposing independent and easier to solve), formal theory and task structure. The last two characteristics correspond to the level of definedness of the domain and the task in [104].

Problem solving processes and strategies. Schemata, procedures and heuristics are types of problem-solving strategies [52]. Here, the word strategy is used as umbrella term. A problem solving strategy is a technique that may not guarantee a solution, but serves as a guide in the problem solving process [91]. Gick described a simplified problem solving process consisting of two sub-processes [52]:

1. generation of a problem representation or solution space (the problem solvers view of the problem);

2. a solutions process that involves a search through the solution space.

During the construction of the *problem representation*, certain features of the problem may activate knowledge in memory. The problem representation contains for example the start state including the obstacles, the goal state including criteria for when the goal is reached, and a description of prior declarative knowledge, rules, and strategies that will aid in solution. A schema for that type of problem may then be activated. Here Jonasson's typology plays an important role, i.e. the problem solver has to recognize the type of the problem and chooses an appropriate schema.

A *schema* is defined as a cluster of knowledge related to a problem type. It contains information about the typical problem goal, constraints, and the solution procedures [52]. The solution procedure, also called procedural rules, is a generalizable series of steps initiated in response to a particular class of circumstances to reach a specific goal. Procedures are often strictly defined (i.e. all steps are included with no ambiguity in each step). A procedure can be an algorithm, simple (a linear sequence of steps), or very complex (with many decision points, at which a solver must determine which of two or more situations exist and based on this determination which branch of the procedure will be followed) [137]. We notice that according to this definition a rule of thumb is a type of procedure too. In contrast with an algorithm, a rule of thumb is loosely defined and there is often ambiguity in a step.

If schema activation occurs, the solver can apply the solution strategies, for example an algorithm and/or a heuristic, contained in the schema. Schema driven strategies may be general or domain specific. An example of a general strategy is decomposition of the problem. An example of a domain specific strategy is modeling an XML content model.

If there is no appropriate schema activation, the problem solver proceeds to the second step and a *search strategy* is invoked. People often use heuristics for searching the problem space. A *heuristic* is a rule of thumb, also

called a weak method, that will generally get one at the correct solution, does not guarantee the correct solution, and are independent of a particular problem [43, 96]. Examples of heuristics are:

- Randomly pick up a reachable next state (used when we have no idea how to reach the goal state) [43];
- Hill climbing (move repeatedly to the state that looks most like the goal state) [43];
- Problem decomposition (breaking the problem into sub-problems) [43, 51];
- Mean-ends-analysis (where the difference between the current state and the goal state is (recursively) decomposed) [43, 51],
- Reason by analogy (if the solver has solved a similar problem in the past, the solver can go directly to the solution by mapping the solution of the old problem onto the current problem) [43, 128].

Design and modeling. Jeffries et al. (1981) presented a theory of the global process that experts use to control the development of a software design [69]. They stated that designing a software system is a complex task and can not be solved by a single, well-understood schema. Instead, the major task is a recursive reduction of the original, ill-structured problem into a collection of more-or-less well-structured problems. The design task is directed by a design-schema which consists of abstract knowledge about the structure of a completed design and the procedures involved in the generation of that design. The schema is assumed to include:

1. a collection of components that partition the given problem into a set of meaningful tasks;
2. components that add elements to tasks in order to assure that they will function properly (e.g. initialization of data structures);
3. a set of procedures that control the generation of designs;
4. evaluation and generation procedures that ensure effective utilization of knowledge.

The major control processes of the design-schema are summarized into a set of abstract production rules. These rules are an attempt to capture the global control process only, i.e. many aspects are not addressed at all. Notice

that this description does not consist of one schema and one procedure. Instead, this description consists of several (sub-) schemata and processes.

The design-schema allows several decomposition methods. The design-schema is presented using the input-process-output strategy. Using this strategy, the initial model is specified by a set of tasks and a control structure that will solve the problem. Each ill-structured task is then recursively expanded into a set of well-structured tasks or problems. The final solutions of these sub-problems represent a solution to the original design problem.

Other decomposition approaches can be used. Examples are data-structure-oriented approaches (Jacksons (1975), Warnier (1974)), the data-flow oriented approaches (Myers (1975), Yourdan (1975)) and the more modern object-oriented approaches using the UML-notation [85].

Educational considerations Merriënboer and Kirschner have presented a four-component instructional design model for the learning of complex constituent skills [96]. They argue that many constituent skills, as modeling is, can only be performed if the learner has integrated declarative and procedural knowledge related to the problem domain. These types of knowledge are needed to solve a task, for example to study a case-study, to carry out a project, to solve a problem, or to make a model of a certain situation.

They distinguish two types of constituent skills, namely schema-based and rule-based skills. *Schema-based* or *non-recurrent skills* are controlled, schema-based processes that are performed in a *variable* way from problem situation to problem situation. These schema-based skills involve the *different* use of the *same* knowledge in new problem situations. Developing a domain model is an example of a schema-based skill. A schema consists of domain knowledge (in the form of mental models) and guiding knowledge (in the form of cognitive strategies). For schema-based skills, supportive information is important. Supportive information provides learners with the givens, the goals, and solutions that get them from the givens to the goals. It explains how a domain is organized in terms of mental models (i.e. conceptual, structural, and causal models) and how to approach problems using these mental models (cognitive strategies in terms of systematic approach to problem-solving (SAP)) in that domain.

On the other hand, *rule-based* or *recurrent skills* are processes that are performed in a highly consistent way from problem situation to problem situation. Developing an precise XML content model is an example of a rule-based skill. For rule-based skills procedural information is important. Procedural information describes how a problem is solved step-by-step.

Anderson (1993) has developed the Adaptive Control of Thought theory

(ACT) [3]. This theory is currently the most comprehensive theory describing the learning processes responsible for the *automation of rules*, also called *production rules* [96]. An example of a production rule in the context of geometry proofs is:

IF	the goal is to prove two triangles are congruent
THEN	set as a sub-goal to prove that all six pairs of sides and angles are congruent.

Based on the ACT theory, Anderson et al. have developed cognitive tutors for practicing problem solving in LISP, geometry and algebra and extracted eight principles [4, 5]. We briefly list the five principles for as they are relevant for practicing modeling skills:

- The first principle is *Represent student competence as a production set*, which means that the tutoring system should be informed by an accurate model of the target skill and very precise about the instructional objectives. The ACT theory says that the model should be cast as a production system: which rules may be applied in which order. Only then the system is able to follow the student's problem solving process, can detect erroneous rule applications and missing rules, and can correct erroneous steps made by a student. In cases the production system consists only of high level or heuristic rules the model can only test on these high level steps and the required qualities of the final model or sub-models.

- The second principle *Communicate the goal structure underlying the problem solving* emphasizes the importance of knowing the goal structure as part of a problem solving process. Solving a problem means decomposing the problem into a hierarchical structure of goals and sub-goals. This principle says that exposing and communicating these goals should be an instructional objective.

- The sixth principle is *Provide immediate feedback on errors.* Anderson et al. argue that making errors can severely add to the amount of time required for learning, can demotivate the learner, and that it is difficult for learners to learn the correct production from an episode involving applying the wrong productions. The ACT theory predicts best learning if students are told immediately why they are wrong and what the correct actions are.

- The seventh principle *Adjust the grain size of instruction with learning* means that it should be possible to process the student's problem

solving in larger units of analysis, i.e. it should be possible, if the student's experience increases, a student can increase the grain size with which a problem is solved.

- The eighth principle *Facilitate successive approximations to the target skill*, means that in the initial stages of the learning process the student can not solve all the solving steps. The tutor can fill in the missing steps. During later stages in the learning process the student will be providing more and more of the work until the tutor is completely in the background.

The other principles are less important in our context, because they are related to tutoring and less to practicing. These principles are: *Provide instructions in the problems-solving context*, which means providing instruction between each new section, where a section begins where new production rules are introduced, *Promote an abstract understanding of the problem solving knowledge*, which means students should be prevented from developing only specific knowledge from particular problem-solving examples and should be stimulated to develop more general applicable production rules, *Minimize working memory load*, which means minimizing presentation and processing of information that is not relevant to the target productions.

Beeson [16] has listed eight criteria that must be met if we are to provide successful computer support for education in algebra, trig, and calculus. The first two are *cognitive fidelity*, which means that the software solves the problem in the same way as the student should solve it, and the *glass box principle*, which means that a student can see how the computer solves the problem. The third one is the *customised to the level of the user* criterion, which means that the system should be adaptable to the level of the student. The fourth criterion is the *correctness principle*, which means that a student cannot perform incorrect operations. The fifth criterion, *user in control*, means that the student decides what steps should be taken and the computer can help a student when he or she is stuck. The sixth criterion is *the computer can take over if the user is lost*, which means that when a student is stuck the system can produce a next step. The seventh criterion is *easy to use*, which means for example that no unnecessary typing is required and clear feedback is produced. The eight criterion is *usable with a standard curriculum* which means that a standard curriculum in mathematics should be supported.

Feedback. As stated in the previous chapter (section 1.4) there is still a debate about the best timing of feedback, immediate versus delayed. The

same is true about the form and content of feedback messages, i.e. lessons and learners vary so greatly that it may not be possible to specify a systematic relation between feedback content and instructional performance prior to the deliverance of the lessons itself [83]. That feedback has a positive effect is a fact [83]. Furthermore, it seems there exists an agreement that immediate feedback produces a better effect during initial stages of learning [58, 113].

Although the ultimate goal of learning problem solving is the ability to perform the solving schema as a whole, it is helpful to firstly practice the components separately before moving to a complete schema. Smith and Ragan (1993) have described some practices for learning procedures [137] and the type of feedback convenient for each type of practice:

- Learning to determine if the procedure is required. For this kind of practice simple correct/incorrect feedback can be given eventually supplemented with explanations about the (in)correctness.

- Learning to complete the steps in a procedure. For this type of practice, learners should be informed about the correctness of each step and the correctness of the decisions made on each decision point in the procedure. Furthermore, the feedback message should contain qualitative information as to whether the inputs into the operation were appropriately selected, whether the outputs of the operation reached any prescribed criterion, and whether the step was completed with acceptable precision and efficiency. For some procedures, as for example for some mathematical operations, this may be fairly straightforward. For other procedures, as for example heuristics, this is difficult.

- Learning to list the steps in a procedure. For this type of practice information about whether all steps are executed in the correct order may be included in the feedback in response to the entire procedure.

- Learning to check the appropriateness of a completed procedure. For this type of practice, correct answer feedback is given followed by a detailed explanation of whether and why the procedure was (in)correctly completed.

Specifically for solving more complex problems, Smith and Ragan mention that practice may initially include instructional guidance such as hints, guiding questions, presentation of the rules, suggestions for strategies and information about the efficiency of the solution process. During initial stages of practice, feedback should be immediately available for intermediate stages.

Sales (1993) listed seven different roles of feedback in technology-assisted instruction [132]. These roles are:

- *Direct* – provides information about an action to be taken ('Apply left factoring').

- *Inform* – acknowledge the accuracy of learner's input ('Correct').

- *Instruct* – supplemental information intended to improve the learner's understanding ('Incorrect. You have tried to apply the left factor rule, but a bracket is missing. Try again!').

- *Motivate* – provide a reward or incentive designed to create a motive for continued effort (a score increased after each correct step).

- *Stimulate* – arouse the learner to continue (after a period of inactivity, a beep from the computer encourages the student for input).

- *Advise* – alerts the learner to the status of his or her efforts in relation to the required criteria ('You must add attributes to the classes before finishing the exercise').

- *Summarize* – a cumulative report of the learner's performance ('Three out of four content models were correct').

2.5 Related work on modeling ITSs

As far as we know, ITSs for practicing modeling which produce semantically rich feedback are not wide spread. Other authors report this observation too [13, 111, 116].

We will discuss a number of ITSs for well-defined as well as for ill-defined domain/task combinations. Generally, five ITS-strategies can be distinguished [88]:

- *Model-based* – The student's solution is compared to an author's solution model. The author solution model can describe the final model as well as the solution process.

- *Constraint-based* – The student's model is compared to a set of constraints. The constraints describe especially the final model.

- *Discovery learning* – A student learns domain knowledge or declarative knowledge by discovery.

- *Case analysis* – A student learns by examination of past cases.

- *Collaboration* – Students learn by solving problems in collaboration in which negotiation plays an important role.

For this research about feedback during practicing modeling activities, the first two strategies are of interest. Discovery learning concentrates especially on learning declarative knowledge. Case analysis is not a model activity but, in cases of modeling education, an evaluation of a created model (which could be in some respect an interesting exercise). Collaboration does not have our attention in this research. COLER is an example of a collaborative environment for entity relationship modeling [36]. A general review on related work can be found in [111].

2.5.1 Model-based

In the model-based approach, a student's solution model is compared with one or more author's solution models. These authors' solution models represent one or more acceptable solution models to a given analysis or design problem, or a general model of the domain as a whole. A solution model describes the final model solution. The solution model can describe the strategy to reach the final model as well. The model is used to check the student's actions, to provide help, as well as to test the student's final model [88].

We can distinguish between strong model-based tutors and weak model-based tutors [88]. *Strong model-based* tutors require strict adherence to their contents. These tutors have been proven successful in well-defined domains such as mathematics, but not in ill-defined domains as for example modeling and design. Being successful in ill-defined domains requires a formalization of the domain, i.e. a well-defined subset of the domain is created. On the other hand, *weak model-based* tutors use the model as a guide but do not require strict adherence to their contents.

Generally, the model-based approach has three problems, which are still not solved [144]:

- generations of sample solutions may be too expensive, if there are many different ways of solving the problem;

- checking the correctness of a correct student's solution which does not match one of the sample solutions completely is difficult, and

- generating feedback for a partly finished solution matching several sample solutions is difficult.

These problems are the reason why the model-based approach is particular usable in well-defined domains. We will describe two model-based tutors, namely ERM and DesignFirst-ITS. Other examples of model-based approaches are described by Gross et al. [55], an ITS based on clustering solution spaces using machine learning techniques, and Schramm et al. [134], who describe a system which uses an author's solution and compares the expert's solution and the student's solution based on the number of classes, attributes, etcetera.

ERM-VLE. Entity Relationship Modeling (ERM) [57] is a text based environment for learning Entity Relationship (ER) modeling. Students construct an ER model from a written requirement description by moving through a virtual space consisting of a number of rooms for creating entities, relations between these entities and attributes. The modeling task depends on the textual analysis of a given scenario, where nouns correspond to entity types or attributes and verbs correspond to relationship types. The topological organization of the virtual world corresponds to the task structure, i.e. first entities are determined, then all relationships are determined, and finally attributes are assigned to the entity types.

The environment provides immediate feedback on syntax, semantic as well as on methodological errors. For feedback on the semantics of the model, the solution for the scenario is embedded (hard coded) in the virtual world, i.e. the correspondences between the phrases in the scenario and the entity types, relationship types and attributes in the ER model are stored. Feedback on the methodology is given on the basis of the topological organization of the virtual space, which reflects the task structure.

Disadvantages of the environment are the hard coded implementation of the solution to a certain requirement description and the hard coded task structure which is only applicable to ER-modeling. Furthermore, the student is only allowed to establish the system's ideal correspondences, i.e. the author's solution. Often, there are more solutions possible.

A preliminary study has been performed, which suggests that the system is easy to use and effective with learners.

DesignFirst-ITS. In DesignFirst-ITS [112], linguistic analysis is used as the process for translating a problem description into an OO-class diagram. An author enters a problem description. A solution generator automatically generates class diagrams as potential solutions. An instructor tool displays the potential solutions and the author can revise the solutions to his or her preferences.

Students design classes, methods and attributes in a separate editor. For matching a student's component name to the names in the solution space, a string-matching algorithm is used. Abbreviations and spelling errors are considered by applying string similarity metrics. An expert evaluator evaluates each of the student's steps by comparing it with the solution template [111]. Each deviation from the standard solution leads to a feedback message, which are authored by the author.

We have doubts about the effectiveness of the way alternative correct solutions are matched to the standard solution and to which extent more complex diagrams are correctly evaluated. The expert evaluator described in [111] performs only a string matching algorithm and does not perform any analysis on structural aspects as for example associations and multiplicities.

The authors report positive results, namely an overall accuracy rate of 87 percent. Teachers as well as students are positive about the ITS.

IDEAS. Another set of interesting tools are developed in the project IDEAS (Interactive Domain Reasoners)[3] [63]. IDEAS is a framework for developing domain reasoners that give intelligent feedback. In each domain reasoner, the domain knowledge as well as the procedural knowledge are described in great detail. Domain reasoners have been developed for example for solving linear, quadratic as well as higher-degree equations, Gaussian elimination, and rewriting logical formulas into disjunctive normal form. In these tools, the domain knowledge is a mathematical language such as a system of linear equations. The procedural knowledge is a fine-grained description of all valid sequences in which a rewrite rule from a certain set of rewrite rules can be applied. Notice that this level of detail corresponds to our examples about modeling and rewriting precise XML content models in section 1.6.

For specifying these strategies for solving mathematical exercises, a special language is developed. Using this language, worked-out examples can be automatically generated, the progress of a student can be automatically tracked by inspecting submitted intermediate answers, and suggestions can be automatically reported back in case the student deviates from the procedure. As a result, it becomes less labor-intensive and less ad-hoc to specify new domains and exercises within that domain.

Another distinguishing feature of the tools is the detailed feedback that they provide on several levels. Examples are a hint for the next step, an error message, a message about a correct but suboptimal steps, and a worked-out solution to an exercise.

[3]See for more information: http://ideas.cs.uu.nl/www/

2.5.2 Constraint-based

A constraint-based model (CBM) represents requirements as a set of constraints (rules or procedures). The constraints select, out of the set of all possible solutions, all correct solutions [118]. A constraint is often modeled as an ordered pair, namely a relevance condition and a satisfaction condition. The relevance condition identifies the class of problem states for which the constraint is relevant. The satisfaction condition identifies the class of relevant states in which the constraint is satisfied. A constraint can be interpreted as: if the properties of a certain relevance condition hold, then the properties of the corresponding satisfaction condition have to hold too, otherwise something is wrong [118]. Constraints can represent syntactic properties as well as semantic properties. Furthermore, constraints can be used to model problem states and are as such usable to define phases in a solution process [118].

The constraints could specify a complete solution or a partial solution. Constraints can be strong or weak. Strong constraints represents requirements that must be satisfied. Weak constraints represents preferences or warnings [88].

The constraint-based model approach is particularly suitable for ill-defined problems, where we are not able to represent all knowledge needed to solve the problem, i.e. declarative as well as procedural knowledge. Instead, using the constraint-based approach, only a set of constraints on the final solutions have to defined. A correct model solution submitted by the student can be recognized, even if that solution deviates from the ideal solution: if no constraint is violated, then the solution is correct with respect to the notion of correctness embodied in the constraint-base.

A disadvantage of constrained-based tutoring systems is that these tutors do not follow the student step by step and do not give feedback after each individual problem solving step [112]. In [101] the author claims the tutor supports procedural skills, but the procedure described (database normalization) is a sequential number of steps a student follows step by step. Furthermore, this approach cannot easily provide feedback after each solution step or suggest a next step to a student who is stuck [112]. Feedback is given after the student submits his or her (more or less) complete solution. Although this is appropriate for some problem domains, we consider this as inconvenient in the general case of modeling. Furthermore, it is not always a simple task to define all constraints needed. With an incomplete constraint-base, some incorrect solutions might mistakenly classified as correct. Therefore, in some constraint-based tutors, supplementary techniques,

as comparison of the student's solution to an ideal solution, are used.

The Intelligent Computer Tutoring Group from the University of Canterbury (New Zealand) has developed a number of constraint-based tutoring systems. In the next two paragraphs we briefly describe two of these tutors. Other tutors developed by the group are SQL-Tutor [102], NORMIT [101] (a tutor for database normalization), and EER-Tutor [98, 161] (a tutor for database design).

Other examples of constraint-based approaches are described by Striewe and Goedicke [144], a rule-based approach for automated checks on UML diagrams, design critics used in ArgoUML[4], and Fischer et al [46], who have described an architecture for design environments as for example designing a kitchen. They use the word critic in stead of constraint.

KERMIT. KERMIT is an ITS for learning Entity-Relationship (ER) modeling [145]. The system is designed as a complement to classroom teaching. Students construct ER schemata that have to satisfy a given set of requirements. Each exercise is presented as a text describing the requirements of the database that should be modeled. Once the student completes the modeling task or requires support from the system, the system analyses the student's model and gives feedback.

The system requires the student to name each newly added construct by selecting a word or phrase from the problem text. Typing a new name is not possible. In this way, the task of finding correspondences between the student's model and the constraint-base is solved. Another technique used is string-matching [112].

The knowledge base of KERMIT consists of syntactic as well as semantic constraints. The syntactic constraints describe the syntactically valid ER schemata. The semantic constraints compare the student's solution to the ideal one. Depending on the number of constructs that violated a constraint, a simple feedback messages or a more detailed one is presented to the student.

The system gives feedback using an animated agent and text boxes. The animated agent (Genie) presents instructional messages using animation. The student can request for more detailed feedback messages. Feedback is offered at five levels of detail, namely simple feedback (correct/incorrect), error flag (which type of construct contains the error), hint (a general feedback message from the first violated constraint), detailed hint (a more detailed feedback message), all errors (a list of all feedback messages according to vi-

[4]See: http://argouml.tigris.org.

olated constraints), and solution (the full ER diagram). As a student starts with modeling, the feedback messages are set to simple. The level of feedback is incremented with each submission until the level reaches the detailed hint level. The system observes students' actions and adapts to their knowledge and learning abilities.

The system is evaluated in two evaluation studies. The system proved to be successful, i.e. students showed significantly better results in comparison to students who practiced ER modeling conventionally.

COLLECT-UML. COLLECT-UML [12, 13] is a client-server application in which students can practice object-oriented analysis and design using UML. This system too requires the student to name each newly added construct by selecting a word or phrase from the problem text. The system observes students' actions and adapts to their knowledge and learning abilities.

Feedback is offered at five levels of detail, namely simple feedback (correct/incorrect), error flag (which type of construct contains the error), hint (a feedback message from the first violated constraint), all hints (a list of all feedback messages according to all violated constraints), and full solution (the full UML class diagram).

The authors report a significant increase of students' performance.

2.5.3 Authoring

ITSs still have not achieved a widespread effect on education due to their high complexity and difficulty of development. The reason for this is that composing the domain and task knowledge required for an ITS consumes a large amount of labor. As such, most of the ITSs are prototypes, never leave the laboratory stage, and/or are applied in other adjacent domains. Authoring tools can play here a crucial role, because they can simplify the task of composing the domain and task knowledge for an ITS [116].

Murray distinguishes two broad categories of authoring systems, namely pedagogy-oriented and performance-oriented [115]. In the first category, pedagogy-oriented systems, the focus is on how to sequence and teach relatively canned content. Guidance and planning is at a more global level looking at an optimal sequence of topics and prerequisite knowledge. In the second category, performance-oriented systems, the focus is on providing rich learning environments in which students can learn skills by practicing them and receiving feedback. Here, special attention is payed to the representation of human problem solving skills. Performance-oriented systems focus on feedback and guidance at the level of individual skills and procedural

steps. Domain knowledge and procedural descriptions are important parts of these systems. ITSs for practicing modeling fall in the second category.

Generally, authoring tools should support the following six goals [116]:

- decrease the effort for making an ITS,
- decrease the skill threshold for building an ITS,
- help the author to structure the domain and pedagogical knowledge,
- enable rapid prototyping,
- support good design principles, and
- enable evaluation of alternate instructional methods.

Although there has been significant progress in the development of ITS authoring tools and the understanding of the underlying concepts, authoring tools for ITSs are still at the laboratory stage. An overview of actual authoring systems, consisting of twenty-five applications, is presented in [116]. None of these authoring systems are for constraint-based tutors. For constraint-based tutors, two authoring systems have been recently developed. These are WETAS and ASPIRE [103].

Authoring tools which explicitly use ontologies are described by Aroyo et al. [6], a tool based on ontologies supporting the development of domain and task ontologies and performing (semi) automatic course ware authoring activities, and Jin et al. [74], who describe an authoring system that uses ontologies, both domain and task ontologies, to produce feedback (error, warning and suggestion) for an author.

Murray lists several unanswered questions, which must be answered before authoring tools for ITSs are commercially available [116]. One of these questions is to what extent the difficult task of modeling the domain and task knowledge can be supported. Authoring systems make different compromises along the spectrum of free-form design to constrained design. Constrained design restricts authors in using a limit number of templates for expressing domain and task knowledge. These templates ensure consistency, accuracy, etcetera, at the expense of flexibility. On the other hand, more open-ended systems allow for more flexibility. This more added flexibility provided to the author results in a higher probability of inconsistency, inaccuracy, etcetera. One way to allow this flexibility while maintaining quality is to allow the author to enter what she/he wants in the way that she/he wants, but to include mechanisms that check the authored information for consistency, accuracy, completeness, etcetera.

2.6 Research questions

In this thesis we investigate a number of *aspects* of feedback generation during modeling. We have the following research questions:

- How can we produce feedback in a well-defined domain and well-defined task about syntactic mistakes, semantic mistakes, and the (lack of) progress in the modeling process? This question is answered in chapter 3 in which we describe a framework for manipulating mathematical terms. The framework is demonstrated by solving a system of linear equations.
- How can we analyze several properties of a domain ontology, such as completeness and correctness, during the authoring process and thereby allow more flexibility in authoring processes? In chapter 4 we investigate this question by introducing schema analysis as a technique by which we are able to detect a number of possible mistakes that can appear during authoring.
- How can we support different data modeling languages in an ITS without re-developing important parts of the ITS for each new data modeling language introduced? This question is treated in chapter 5.
- What is needed to specify a precise task description, i.e. to transform an ill-defined task into a well-defined task? This question is answered in chapter 6 by developing a precise task description for modeling XML content models.

The goal of this research is to investigate and develop a number of aspects of a general framework for producing feedback in an ITS for practicing modeling. To test our ideas we have implemented prototypes of parts of the framework.

2.7 Haskell preliminaries

In subsequent chapters, we use the functional programming language Haskell for implementing our ideas. We briefly explain Haskell, for more information see [77].

Haskell is a lazy, purely functional, and statically typed programming language with a relatively small core based on the lambda calculus. Laziness implies that Haskell defers the evaluation of expressions until their results are needed. Arguments supplied to a function are only evaluated if their

values are needed. In a functional language, a program is a function that is defined in terms of other functions. It defines what a program should accomplish, rather than describing how to accomplish it. In a functional language, there are no assignments or mutable states. We briefly introduce some constructs we will use in this thesis.

Tuples. The tuple data type `(t1,t2,...,tn)` is constructed from component types. It consists of values `(v1,v2,...,vn)`, in which `v1::t1`, etcetera (where `::` means 'is of type'). Function `fst` selects the first element of a pair, `fst (x,y) = x`, and function `snd` selects the second element.

Lists. We use the data type list extensively. The empty list is denoted by `[]`, and the concatenation of two lists `x` and `y` is denoted by `x++y`. Prepending an element `x` to a list `xs` is denoted by `x:xs`. In a list comprehension `[x | x <- xs, test x]`, a new list is generated from the list `xs`. Each element `x` of `xs` is tested, and, if the test succeeds, added to the new list. Function `map f` takes a list and applies function `f` to all elements in the list, so `map f xs = [f x | x <- xs]`

Functions. Anonymous functions can be constructed using lambda notation, so function `\(x,y,z) -> (x,y)` selects the first two components of a triple. Function `null` tests if a list is empty: `null [] = True`. To check if an element `x` is an element of list `xs`, we use the expression `elem x xs`. Function `zip` takes two lists and returns a list of corresponding pairs: `zip [1,2] [3,4,5] = [(1,3),(2,4)]`, where extra elements in the longer list are discarded. Functions `head` and `tail` extract the first and the remaining elements of a nonempty list, respectively. Function `inits` returns the list of initial segments of its argument list: `inits "abc"` results in `["","a","ab","abc"]`, and function `tails` returns the list of all tail segments of its argument list: `tails "abc"` results in `["abc","bc","c",""]`.

Function composition composes two functions: the output of the second function (`g`) becomes the input of the first function (`f`): `(f.g) x = f (g x)`. The type of a function `f::t1 -> t2 -> t3` can be read as: function `f` takes two arguments of types `t1` and `t2` and returns a value of type `t3`.

The result of a function can be a function. Suppose for example a function for adding up two numbers, i.e. `add x y = x + y`. Using function `add`, we can define a function `successor` as `successor = add 1`. Function `add 1` is partially parametrized and needs one more parameter.

Functions can be passed as parameters. For example, in `map isEven [1,2,3,4]` the instantiated type of `map` is `map::(Int -> Bool) -> [Int]`

`-> [Bool]`. Choice between conditions is represented by a vertical bar `|`. For example:

```
max x y | x >=y      = x
        | otherwise = y
```

means: if the guard `x >= y` is true then return `x`, otherwise return `y`.

Some functions can be defined in a simple and intuitive way using *pattern matching*, in which a sequence of syntactic expressions called *patterns* is used to choose between a sequence of results of some type. If the first pattern matches, then the first result is chosen, if the second pattern matches, then the second result is chosen, etcetera. For example, the binary function `not` can be defined as follows:

```
not :: Bool -> Bool
not False = True
not True = False
```

Functions with more than one parameter can also be defined using pattern matching. Furthermore, the wildcard pattern can be used for simplifying function definitions. An example with two parameters and the use of wildcards is the following definition of the logical function `and`:

```
and :: Bool -> Bool
and True True = True
and _ _       = False
```

Pattern matching can be used for lists and tuples, for example:

```
fst :: (a,b) -> a
fst (x,_) = x

head :: [a] -> a
head (x:_) -> x
```

Chapter 3

Feedback in an ITS for solving equations

3.1 Introduction

In this chapter, we start with a well-defined task, namely solving a system of n linear equations with n variables. Solving a system of equations amounts to rewrite the system in a certain form. Rewriting is a type of model transformation and assumes a start model, in this case a system of equations to solve. These equations could represent an operational problem (the domain of interest), as for example a stock problem as part of a supply chain optimization.

Mathematics is constructive in nature: mathematics students learn to *construct* solutions to mathematical problems. Solving mathematical problems is often done with pen and paper, but an ITS can offer great possibilities. An interactive ITS that support learning mathematics should provide the capability to give feedback to a student at each step. To illustrate our approach, we have built an ITS for solving a system of linear equations. We call this tool the Equation Solver. Figure 3.1 shows a screen shot of the tool.

The Equation solver. The Equation Solver consists of three text fields. The top text field is the working area, in which a student can edit a system

This chapter is based on: Passier, H. and Jeuring, J. (2006). Feedback in an interactive equation solver. In M. Seppälä, S. Xambo, O. Caprotti, editors, Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT, pages 53–68, Oy WebALT Inc. [126]

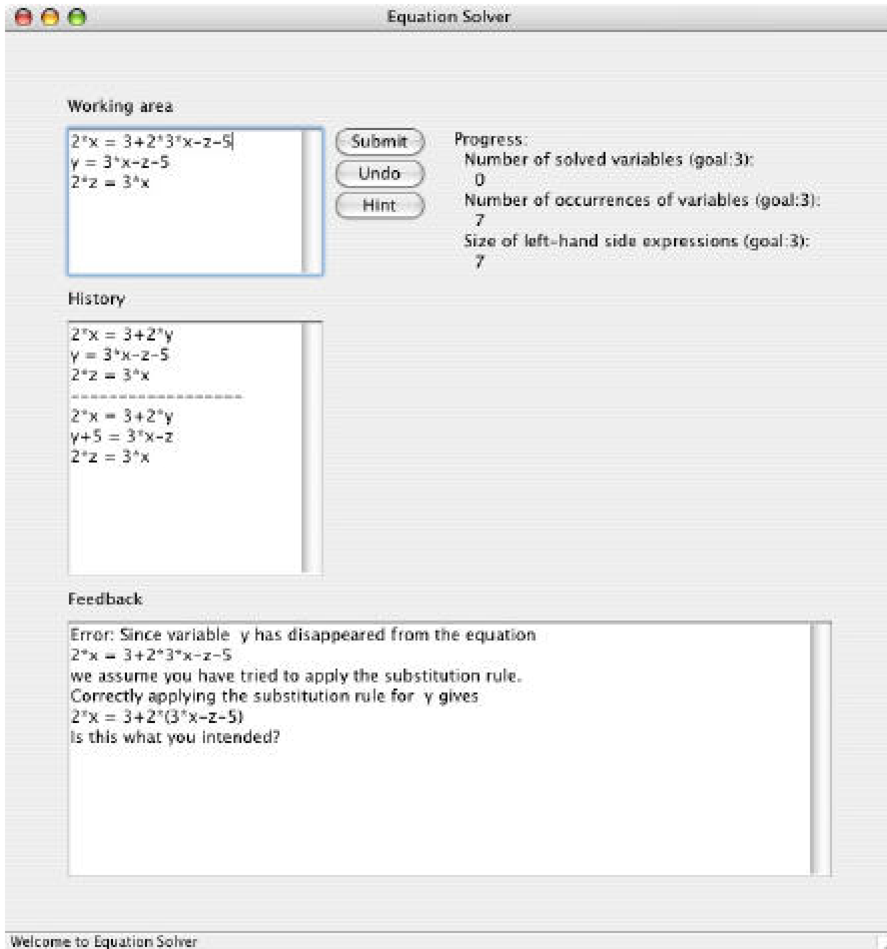


Figure 3.1: The Equation Solver

of equations stepwise to a solution. The current system of equations is

$$\begin{cases} 2 \cdot x & = 3 + 2 \cdot 3 \cdot x - z - 5 \\ y & = 3 \cdot x - z - 5 \\ 2 \cdot z & = 3 \cdot x \end{cases}$$

The second text field displays the history of equations. Apparently the previous system of equations was

$$\begin{cases} 2 \cdot x & = 3 + 2 \cdot y \\ y & = 3 \cdot x - z - 5 \\ 2 \cdot z & = 3 \cdot x \end{cases}$$

and the student replaced y by $3 \cdot x - z - 5$, forgetting to parenthesize the result. The third text field displays the feedback. In the figure it explains why the last step is incorrect.

The Equation Solver presents a system of equations to a student, for example

$$\begin{cases} 2 \cdot x & = 3 + 2 \cdot y \\ y + 5 & = 3 \cdot x - z \\ 2 \cdot z & = 3 \cdot x \end{cases}$$

and the student has to rewrite these equations into a form with a variable to the left of the equals symbol, and a constant to the right of the equals symbol, for example

$$\begin{cases} x & = 7 \\ y & = 11/2 \\ z & = 21/2 \end{cases}$$

The student presses the Submit button to submit an edited system of equations, and the Undo button to undo the last step (or any amount of steps). If a student wants help, he or she presses the Hint button to get a suggestion about how to proceed. Finally, the Equation Solver gives information about progress towards a solution by showing how many variables have been solved, and several other kinds of information.

Feedback in the Equation Solver. The Equation Solver gives feedback about two kinds of mistakes:

- syntactic mistakes, for example when a student writes $y + 5 = 3 \cdot x-$ instead of $y + 5 = 3 \cdot x - z$,
- semantic mistakes, usually mistakes in applying a step towards a solution, for example when a student rewrites $y = 3 + 1$ by $y = 5$,

and it gives feedback about (lack of) progress towards a solution.

The comparison function of the Equation Solver consists of a *solver*, which performs symbolic calculations, an *analyzer*, which analyzes the submitted equations of a student based on a set of rewrite rules for the domain of a system of equations, and several *indicators*, which indicate the progression of a (series of) rewrite step(s).

Note that we try to mimic the pen-and-paper situation as closely as possible, by letting students enter and rewrite equations in a text field. Another approach is to offer the possible rewrite steps to the student, and let the student select a rewrite step, which is then applied to the system of equations by the Equation Solver, as proposed by Beeson [16] in MathXpert. In such a situation, it is impossible to make a syntactic mistake, or to rewrite an equation incorrectly. The former approach has the advantage that a student also learns to enter correct equations, and to choose and apply rewrite steps correctly. Furthermore, it is closer to the pen-and-paper situation. The latter approach has the advantage that a student can concentrate solely on solving a system of equations. The only feedback that needs to be provided in the latter approach is feedback about progress. Although we do not support the latter approach, it is orthogonal to our approach, and easily added to our tool.

Contributions. In this chapter, we discuss the framework for providing feedback, in which feedback about syntactic mistakes, semantic mistakes, and (lack of) progress in the solving process is produced. The framework assumes a well-defined or structured domain (like linear equations), for which a set of rewrite rules (or transformations) is defined (like $x+0 = x$ for all x), a goal is specified (like rewrite all equations to a form where there is a single variable to the left, and a constant to the right of the equality symbol), and one or more measures can be defined with which we can (possibly partly) determine the distance to the goal.

The main results of our work are:

- We show how results from Computer Science, in particular from the term-rewriting and compiler technology (and in particular parsing) fields, can be used to develop tools that provide semantically rich feedback to students.

- We show how using structural information in data for feedback improves the feedback a tool can give.

We think our framework is useful for several purposes. Developing a tool in our framework forces the developer (a lecturer) to be explicit about *all*

aspects of a particular domain, and it helps developers of ITSs to set up a well-structured feedback component that gives better feedback than existing tools.

In almost all electronic learning environments we know of, feedback is hard coded and/or specified separately for each exercise. Including detailed feedback for exercises is thus very labor intensive. Our framework produces feedback for a whole class of problems. In case of the Equation Solver feedback is automatically generated for all exercises belonging to the class of solving linear equations. Another advantage of our framework is that feedback is produced on the level of rewrite steps the student performs when he/she solves an exercise, instead of feedback on the final result of the solving process [59]. This is important in cases where different solving methods can be used and solving methods consist of several rewrite steps.

Schemata. To solve a system of linear equations, secondary school students often learn two strategies. In the first one, called the *substitution method*, each variable is expressed in one or more other variables. For example, in the system of equations:

$$\begin{cases} x + y & = 4 \\ 2 \cdot x + y & = 9 \end{cases}$$

the variable y in the second equation can be replaced by $4 - x$ resulting in the system

$$\begin{cases} y = 4 - x \\ 2 \cdot x + (4 - x) = 9 \end{cases}$$

After a finite number of such steps the system can be solved. In the second method, the *combination method*, one equation is subtracted from another equation. If for example the first equation is subtracted from the second one, we obtain the system:

$$\begin{cases} x + y & = 4 \\ x & = 5 \end{cases}$$

Again, after a finite number of such steps the system can be solved.

A *schema* how to solve this type of problems, consists of a domain description, i.e. systems of linear equations, a set of rewrite rules and an strategy, in this case an algorithm, in which the rules are applied in a certain order to solve the problem.

Of course the schema used in the equation solver should correspond to the schemata explained by the lecturer, so that feedback and hints correspond to

these schemata. The framework itself is independent of a solving strategy: these strategies can be added to the Equation Solver. In the present Equation Solver, the rewrite rules of the substitution method are implemented and feedback is given about the correct application of these rules. The order in which a student applies the rewrite rules is not analyzed. This has been implemented in later versions of the tool (see sections 3.7 and 3.8).

3.2 The feedback framework

Our framework for providing feedback assumes we have the following components:

1. A domain with a semantics.
2. A set of rewrite rules for the domain.
3. A goal that can be reached by applying the rewrite rules in a certain order.
4. A set of progress indicators to determine the distance between the goal and the current situation.

Our framework provides feedback about syntactic errors, semantic errors (incorrectly applied rewrite rules), and about progress, using the progress indicators.

To illustrate our framework, we will use the Equation Solver introduced in the introduction. Solving a system of n linear equations with n variables $x_1, \ldots, x_n$ amounts to finding constants $c_1, \ldots, c_n$ such that

$$\begin{cases} x_1 = c_1 \\ \vdots \\ x_n = c_n \end{cases}$$

is a solution to the system of equations. We describe the components of our framework for the Equation Solver.

Domain and semantics of the Equation Solver. The domain of the Equation Solver consists of a system of linear equations. The top-level type is a list of equations:

```
type Equations = [Equation]
```

Each equation consists of a left and a right hand expression separated by a '=' (in Haskell denoted by the infix constructor `:=:`) symbol.

```
data Equation = Expr :=: Expr
```

An expression is either a constant, a variable, or two expressions separated by an operator '+', '−', '·', or '/'.

```
data Expr = Con Rational
          | Var String
          | Expr :+: Expr
          | Expr :-: Expr
          | Expr :*: Expr
          | Expr :/: Expr
```

The semantics describe how the domain should be interpreted. For the Equation Solver, the semantics are the solution to the system of linear equations.

Rewrite rules for the Equation Solver. A domain has a set of rewrite rules with which terms in the domain can be rewritten. A rewrite rule rewrites a term of a particular domain to another term of that domain. For example, we have the following rewrite rule for expressions: $(a + b) \cdot c \rightarrow a \cdot c + b \cdot c$, which says that we can rewrite the expression $(a + b) \cdot c$ to the expression $a \cdot c + b \cdot c$ (distribute multiplication over addition) in any context in which this expression appears. For a general introduction to rewrite systems, see Dershowitz et al. [40].

Using rewrite rules, we rewrite terms in the domain to some desired form. For the Equation Solver, the goal is to rewrite the given system of equations to a solution. We now informally present the rewrite rules for the domain of the Equation Solver.

We follow the data representation of the domain and distinguish between rules on the level of a system of linear equations, an equation, and an expression. In these rules a, b, and c are rational numbers, x, y, and z are variables, and e is an expression.

- *System of linear equations.* For a system of linear equations we have a single rewrite rule: substitution. If we have an equation $x = e_1$, we may replace occurrences of x in another equation E_i by e_1. Informally:

$$\begin{cases} x = e_1 \\ E_2 \\ \vdots \\ E_n \end{cases} \rightarrow \begin{cases} x = e_1 \\ E_2(e_1/x) \\ \vdots \\ E_n(e_1/x) \end{cases}$$

- *Equation.* For an equation we have four rewrite rules:

  $$e_1 = e_2 \quad \to \quad e_1 \oplus e = e_2 \oplus e$$

  where $\oplus$ may be any of $+$, $-$, $*$, or $/$.

- *Expression.* For an expression we have a large number of rewrite rules. First, constants may be added, multiplied, etc:

  $$a \oplus b \quad \to c,$$

  where c is the rational number sum of a and b if $\oplus$ is $+$, and similarly for $-$, $\cdot$, and $/$. Coefficients of the same variable are summed using the inverse rule of distributing multiplication over addition.

  $$a * x + b * x \quad \to \quad (a + b) * x$$

  Furthermore, multiplication (division) distributes over addition (subtraction):

  $$(e_1 \oplus e_2) \otimes e_3 \quad \to \quad e_1 \otimes e_3 \ \oplus \ e_2 \otimes e_3,$$

  where $\oplus$ may be $+$ or $-$, and $\otimes$ may be $\cdot$ or $/$.

As argued by Beeson [16], many mathematical operations cannot be expressed by rewrite rules, because they take an arbitrary number of arguments and because other arguments can come in between. Moreover, associativity and commutativity cause problems in rewrite rules. Hence, applying rewrite rules or recognizing applications of rewrite rules in user-supplied equations is not a trivial application of pattern matching, but requires more sophisticated programs.

A *normal form* of a term in the domain of a term-rewriting system is a term which cannot be rewritten anymore. The solution of a system of linear equations is not a normal form of a system of linear equations, because, for example, we can always add terms to a term and immediately subtract the same terms. A term-rewriting system *terminates* if for every term t, we can only rewrite t a finite number of steps. Since we can distribute multiplication over addition and vice versa, our term-rewriting system is clearly not terminating. A term t' is *reachable* from a term t, if there exists a sequence of term-rewriting steps with which we can rewrite t into t'. Clearly, given a solvable system of linear equations, the solution of this system is reachable. In a situation in which terms have normal forms, and the rewriting system is terminating it is much easier to give useful feedback, but for most domains about which we want to give feedback these properties do not hold.

The goal of the Equation Solver. The goal of the Equation Solver is to find constants $c_1, \ldots, c_n$ such that $x_1 = c_1, \ldots x_n = c_n$ is a solution to the system of equations. We assume that all systems of equations set as exercises by the Equation Solver are solvable, a property that is easily verified. The goal is reachable by applying the rewrite rules to the system of equations in a certain order.

Progress indicators for the Equation Solver. To inform a student about the progress in solving a problem, we have defined indicators. An indicator is a measure which (partly) describes the distance from the current system of equations to the solution (the goal). There are several ways to indicate the distance between the currents system of equations and the solution. A possibility is to determine the minimum number of rewriting steps needed to rewrite the current system of equations to the solution. In this chapter we investigate indicators that follow the structure of the data. Thus we can provide more specific feedback than just about the distance to the final solution. We have indicators that indicate progress on the level of a system of equations, on the level of a single equation, and on the level of an expression.

In the next sections, we describe how we provide feedback about syntactic errors, semantic errors, and about progress using this frame work.

3.3 Syntax analysis

A student enters an expression in a text field in the Equation Solver. We have to parse this expression in order to analyze it. We use error recovery parser combinators [146] to collect as many errors as possible (not just the first), and to suggest possible solutions to the errors we encounter. For example, when a student enters

$$\begin{cases} 2 \cdot x & = 3 + 2 \cdot y \\ y + 5 & = 3 \cdot x - \\ 2 \cdot z & = 3 \cdot x \end{cases}$$

the tool reports an error, and says it expects a lower case identifier or an integer in the equation $y + 5 = 3 \cdot x -$. Furthermore, it proceeds with parsing $2 \cdot z = 3 \cdot x$, assuming the expression $y + 5 = 3 \cdot x - \langle \textit{identifier} \rangle$ has been entered. The parser combinators are very similar to the context-free grammar for the domain of equations. We have tuned the parser such that common errors, such as writing $2y$ for $2 \cdot y$, are automatically repaired (and reported).

After parsing we perform several syntactic checks, such that the set of variables that occurs in the system of equations has not changed, and that all equations are still linear equations, and not for example quadratic equations, which would happen if the student would multiply both sides of the equation $2 \cdot z = 3 \cdot x$ by x. If such an error occurs, it is reported.

3.4 Rewriting terms

When a student submits a system of equations to the Equation Solver, the analyzer checks if something has changed. If something has changed, the solver checks that the submitted system of equations has the same solution as the original system, and the analyzer tries to infer the rewrite rule applied by the student. Because we do not analyze the order of rewrite steps it is not necessary to determine the rewrite rule that has been applied in the case the solution of the system has not changed, but it might still be useful for the student to see the name of the applied rule. In the case the order of the rewrite steps is analyzed, it is, of course, necessary to determine which rule has been applied. This functionality of recognizing a strategy has been implemented in a later version (see section related work).

If the solution *has* changed, the student has made an error, and it is important to try to report the likely cause of the error.

An important assumption (restriction) we apply here is that we assume a student applies only one rewrite rule per submitted system of equations. In practice, this will not always be the case. This functionality of recognizing multiple rewrite rules has been implemented in a later version (see section related work). Furthermore, in this implementation of the tool we assume a student does not go around in (small) circles, for example a repetition of adding a term on both sides of an equation followed by a subtraction of the same term on both sides. In itself, the detection of these small circles is technologically quite simple.

In the rest of this section, we discuss the feedback produced by the Equation Solver by means of examples on each of the three levels of our domain. To determine which rule a student intended to apply, we follow a hierarchical approach.

Determining a rewrite on the system of equations level. The analyzer starts with trying to find out if the student intended to apply a rule on the level of the system of equations: the substitution rule. The analyzer can determine whether or not the substitution rule has been applied by collecting the variables that appear in the different equations. If one variable

has disappeared from the set of variables that appear in an equation a substitution step has been applied. Here we assume that an expression such as $x - x$ is internally represented as 0, so that replacing $x - x$ by 0 does not lead to the false conclusion that substitution has been applied. The internal representation is some normal form of the expressions and equations, where occurrences of the same variable are combined. The normalized form of an expression is an expression of the form $a_1 \cdot x_1 + ... + a_n \cdot x_n + c$, where each variable occurs once, and all constants have been added in a single constant c. The Equation Solver determines which variable has disappeared, and checks that applying the substitution using that variable leads to the submitted expression. If this is not the case, the Equation Solver reports an error, and shows the correct equation that results from the substitution. For example, if the system of equations:

$$\begin{cases} 2 \cdot x + 2 \cdot y & = 6 \\ y & = 4 - 2 \cdot x \end{cases}$$

is rewritten to:

$$\begin{cases} 2 \cdot x + 2 \cdot (4 + 2 \cdot x) & = 6 \\ y & = 4 - 2 \cdot x \end{cases}$$

the analyzer produces the following error message:

```
Error: Since variable y has disappeared from the equation

                  2*x+2*(4+2*x) = 6

we assume you have tried to apply the substitution rule.

Correctly applying the substitution rule for y results in

                  2*x+2*(4-2*x) = 6

Is this what you meant?
```

There are several things to note about this message: it is only about the equation that contains an error, it tells why it thinks a certain rewrite rule has been applied, and it shows how the correct application of that rule looks.

Determining a rewrite on the equation level. If the analyzer has detected a change in the system of equations and in no equation the set

of variables that occur has changed, the analyzer tries to find out if there exists an equation that has been rewritten. An equation has been rewritten if both the left-hand side and the right-hand side expression of an equation have changed. On the level of an equation, four rewrite rules may be applied:

$$e_1 = e_2 \rightarrow e_1 \oplus e = e_2 \oplus e$$

where $\oplus$ may be any of $+$, $-$, $\cdot$, or $/$. The analyzer determines whether or not these rules have been applied by comparing the new equation with the old equation. For example, if the previous system is:

$$\begin{cases} 2 \cdot x + 2 \cdot y & = 5 \\ x - y & = 2 \end{cases}$$

and the submitted system is:

$$\begin{cases} 2 \cdot x + 2 \cdot y & = 5 \\ x - y + y & = 2 + y \end{cases}$$

the analyzer concludes that the rule: $e_1 = e_2 \rightarrow e_1 + e = e_2 + e$ has been applied on the second equation of the system. In general, the analyzer can infer an application of the addition (and subtraction) rule on the level of an equation by calculating the value of the expression $(l - l') - (r - r')$, where $l = r$ is the equation in the previous system, and $l' = r'$ is the submitted equation. If this value equals 0, then it is likely that the student has performed an addition (or subtraction) step with value $l - l'$ on both sides of the equation. If the value equals a constant unequal 0 or a variable (possibly multiplied by a constant), then it is likely that a student has performed an addition step, but has made an error in doing so. This error is reported. Finally, if the value is not a constant or a variable, it is likely that the student has performed a multiplication (or division). To determine if a multiplication step has been performed, the analyzer calculates the value of $(l/l') - (r/r')$. If this value equals 0, then it is likely that the student has performed a multiplication (or division) step with value l/l' on both sides of the equation. If the value equals a constant, then it is likely that a student has performed a multiplication step, but has made an error in doing so. This error is reported. Finally, if the value is not a constant, something serious is wrong.

Determining a rewrite on the expression level. If no rewrite on the level of a system of equations or on the level of an equation has taken place, the analyzer tries to determine if a rewrite on the level of an expression

has taken place. It is easy to determine which expression in the system of equations has been changed.

For example, suppose the previous system is:

$$\begin{cases} 2 \cdot (2 + y) + 2 \cdot y & = 5 \\ x & = 2 + y \end{cases}$$

and the submitted system:

$$\begin{cases} 2 \cdot 2 + 2 \cdot y + 2 \cdot y & = 5 \\ x & = 2 + y \end{cases}$$

The analyzer infers that the left-hand side expression of the first equation has changed. The analyzer checks that the normalized form of the new expression and the previous expression are the same. Furthermore, the analyzer tries to infer which expression rewrite rule has been applied. It does this by determining the expression difference between the old expression and the new expression. The expression difference of two expressions consists of the sub-expressions that have disappeared from the old expression in the new expression, and the sub-expressions that have appeared in the new expression. In the above example, the expression difference is $2 \cdot (2 + y)$ (disappeared) and $2 \cdot 2 + 2 \cdot y$ (appeared). These expressions match the rewrite rule for distributing multiplication over addition. If the normalized form of the new expression and the old expression are different, an error is reported, and the analyzer shows all possible correct rewrites of the sub-expression that has disappeared from the expression.

The hierarchical approach to determining which rewrite rule has been applied allows us to pinpoint precisely, in many cases, which mistake (likely) has been made.

3.5 Progression and indicators

An indicator gives a distance from the current system of equations to the solution (the goal). It is used to inform a student about the progress towards a solution. Before calculating the value of the various indicators, the Equation Solver detects whether or not a student has completed the problem. In that case, the system of equations has the form of $x_1 = c_1, \ldots, x_n = c_n$. This is easily detected.

We have defined a number of indicators in the Equation Solver.

- The first indicator calculates the number of variables for which a student has found a solution. If this number increases the student makes progression.

- The second indicator calculates the number of occurrences of variables in a system of equations. Progression is made if this number decreases. For example, in the system:

$$\begin{cases} 4 + 2 \cdot y + 2 \cdot x & = 5 \\ x & = 2 + y \end{cases}$$

  there are four occurrences of variables. Substituting $2 + y$ for x in the first equation reduces the value of this indicator by one. Sometimes the value of this indicator increases due to a substitution, so we do not enforce that the value of this indicator decreases or stays the same at each step.

- The third indicator checks if the expression size of the left-hand side expression of an equation has decreased. Since in our solution we want the left-hand side expression of an equation to be a single variable, a reduction in the size of the left-hand side expression (without removing all variables, since in the end a single variable should remain) indicates progression. For example, rewriting the expression $y + 3 - 1$ to $y + 2$ reduces the size of the expression from 5 to 3 (where operators, constants, and variables all count for 1).

The indicators are independent of the rewrite rules in the Equation Solver. So if a student performs a transformation on the system of equations that does not change the semantics of the system of equations, but for which the analyzer cannot find a corresponding rewrite rule, the indicators can still inform the student about his or her progress.

3.6 Hints

If a student is stuck, he or she can press the hint button. The Equation Solver will then give a next step, or a hint to help the student to produce a next step. The next step depends on the solving strategy used. We have only implemented the Gaussian solving method in the Equation Solver. The Equation Solver produces a next step or a hint based on the previous system of equations submitted by the student, the set of rewrite rules and the solving strategy. For example, if the previous system submitted by the student is:

$$\begin{cases} 4 + 2 \cdot y + 2 \cdot x & = 5 \\ x & = 2 + y \end{cases}$$

the system will suggest:

```
Try to substitute 2+y for x in the first equation.
```

Various levels of help are possible depending on, for example, the solving method, the tutorial strategy and the maturity of the student. In the above situation, where substitution is used as the solving method, we can think of the following, increasingly detailed, messages:

```
Try to apply the substitution rule.

Try to substitute 2+y for x in the first equation.

Substituting 2+y for x in the first equation results in

4 + 2*y + 2*(2+y) = 5

x = 2 + y
```

Different tutorial strategies can be implemented. Besides the messages showed, the system can present for example only the rule that has to be applied and ask the student to apply this rule in the current system of linear equations, or it presents the rule in the context of a simpler task. The last strategy can be valuable in case of solving systems with more than two equations.

3.7 A general tool

In this chapter, we have made a start with a problem from the category well-defined domain and well-defined task. Although the main ideas behind the analysis for feedback in the Equation Solver are reusable in other domains, the implementation is not reusable. For each rewrite rule we have a separate analysis function. All these functions operate on the domain of equations. When a new rewrite rule is added to the system, a new analysis function has to be implemented. When a new domain together with rewrite rules is specified, we have to build a completely new solver and analyzer. This is labor intensive and requires advanced knowledge and experience.

Therefore we want to implement a general tool (a feedback engine), which takes the domain of interest together with rewrite rules as inputs and automatically transforms the rewrite rules in analysis functions. These functions determine which rewrite rule has been applied if an expression has changed. Furthermore, if an error has been made, the analysis functions determine which rewrite rule was probably applied and calculate the correct solution.

This general tool has been developed and is briefly described in section 3.8.

3.8 Related work

At the time we wrote the underlying paper of this chapter (2006), we found little literature on structured feedback in ITSs, and the way feedback is produced. Due to the IDEAS project, which project is briefly described at the end of this paragraph, this situation has changed.

Most intelligent tutoring systems that have a feedback component use techniques from artificial intelligence to report feedback. We think that using the structure in the data and the rewrite rules, we can give more precise and detailed feedback. Of course, there will still be situations where our feedback is insufficient: the amount of possible errors a student can make, and the misconceptions a student can have is close to infinite.

Within the Galois project [19, 20] a digital environment was created in which secondary school students can practice mathematics and perform tests. One of the goals of the project is to automatically provide intelligent feedback. Intelligent feedback is detailed information given to a student, based on an analysis of the students' answer to a question. Feedback should be based on expert knowledge of the mathematical subject, a model of frequently made mistakes, and knowledge about learning strategies needed to select a suitable feedback form. The authors observe that most of the feedback in electronic environments is primitive and only contains information about the correctness of an answer. They describe how to produce feedback for exercises with a numerical answer. The expected answers are categorized, and provided with feedback. An example of such an exercise is: 'A bicycle tyre has a puncture. Every minute 6 percent of the air escapes from the tyre. Which percentage of air has escaped after 9 minutes?' The expected answers are for example (correct) '43', feedback 'excellent'; '42', feedback 'you are close to the correct answer'; '54', feedback 'apply rule . . . '; 'otherwise' feedback 'incorrect answer'. The expected answers are, besides the correct answer, related to frequently made mistakes and listed by experienced teachers. Feedback is updated automatically if the numbers in such an exercise are changed. An advantage of this approach is the detailed feedback that is given for a certain type of exercise. Disadvantages of the approach are in our opinion the impossibility of reuse for other types of exercises and the lack of feedback on a sequence of solving steps.

Other tools for solving systems of linear equations pay little or no attention to feedback. For example, the Linear System Solver (using determi-

nant) [31] returns 'ERROR in perl script on line 23: Illegal division by zero at (eval 129) line 37', if an unsolvable system of equations is entered. Commercial tools such as Algebrator [139] and MathXpert [16] do give feedback on the syntactic level and hints about making progress, but do not use a structural approach to providing feedback about rewriting steps entered by the student.

Cohen et al. [33] have developed a tool for solving exercises about computing the derivative of elementary functions. The tool uses rewrite rules called domain rules and decomposes the original problem into sub-problems obtaining a multistep exercise based on a solution graph. An interactive exercise is then seen as a collection of problems together with the order in which they are solved. According to the students' answer and a predefined strategy, a next step is selected. The correctness of a students' answer is evaluated by a computer algebra system. In this way, a student is guided in solving the initial exercise. Our approach is not based on a solution graph, but uses indicators to inform the student about progression of the solving process and a rewrite analysis to determine which rewrite rule has been (correctly or incorrectly) applied. As a result, the steps a student can take are not limited by a predefined set of rewrite rules or solving strategy, but can be any combination of correct or erroneous rewrite steps. If a step is erroneous, the tool of course complains, but it also tries to give a helpful error message to the student. If the tool cannot recover which rewrite rules have been used, the indicators can still help a student in determining whether or not he or she is on the right track.

Heck et al. [59] describe a system for diagnostic testing of mathematics students. The system is based on Maple T.A., for automated assessments, and Maple, for verification of the students answers. Classes of exercises can be defined. In the given examples, feedback is just the correct answer together with a short description of how the problem can be solved.

Marvrikis et al. [90] describe a web-based learning environment for studying mathematics. The system contains a feedback mechanism that follows an incremental hinting process that changes according to, for example, the time passed, the current goal, and the amount of help a student requests. A mechanism tracks the goals that the author of the activity sets and a student has to reach. The goals involve, for example, selecting an answer for a multiple choice question, putting objects into certain positions, and giving numerical answers. The feedback mechanism comprises production rules defined by an author. It is not completely clear to us how feedback is generated and to which extent the mechanism is reusable.

IDEAS. The Equation Solver was the first ITS in the project IDEAS[1]. After the Equation Solver, a formalism is developed for specifying strategies for solving exercises stepwise in the domain of well-defined problems. Examples of such strategies are reducing a logical expression to disjunctive normal form (dnf) by first pushing $\neg$'s over $\vee$'s and $\wedge$'s using De Morgan's rule, until they are in front of literals, and then distributing $\wedge$ over $\vee$ [87], and solving a system of linear equations by subtracting equations from top to bottom, and then substituting variables from bottom to top (this chapter).

To give an impression of this language, the following fragment (expressed in Haskell) shows a strategy for reducing a logical expression to dnf [70]:

```
dnf = eliminateNots <*> moveOrToTop

eliminateNots = repeatExhausted
  (basic DeMorganAnd <|> basic DeMorganOr <|> basic NotNot <|>
     basic NotTrue <|> basic NotFalse)

moveOrToTop = repeatExhausted
  (bottomUp (basic AndLeftOverOr <|> basic AndRightOverOr))
```

The symbols `<*>` and `<|>` are combinators. The first one (`<*>`) takes two recognizers, and tries to recognize the first followed by the second. The second one (`<|>`) takes two recognizers too, and tries to either recognize the first or the second. The recognizers `eliminateNots` and `moveOrToTop` are sub-strategies and are specified separately. The `basic`-combinator recognizes a single transformation step. Examples of basic transformation steps, as `DeMorganAnd`, `NotTrue`, and `AndRightOverOr`, are:

$$\begin{array}{rl} \text{DeMorganAnd}: & \neg(x \wedge y) = \neg x \vee \neg y \\ \text{NotTrue}: & \neg true = false \\ \text{AndRightOverOr}: & (x \vee y) \wedge z = (x \wedge z) \vee_(y \wedge z) \end{array}$$

The strategy language is a domain specific embedded language, with a clear separation between context-free and non-context-free parts, in which strategies are specified as context free grammars [63]. Domain specific means specific for the domain of strategies The strategy language is embedded (implemented) in the programming language Haskell. The combinators, as for example `<*>` and `<|>` are context free and can be used in any domain. The rule recognizers, as for example `DeMorganAnd` and `NotTrue`, are bound to a domain as for example the domain of logical expressions.

[1]See: http://ideas.cs.uu.nl/www/

A strategy as `dnf` recognizes sentences consisting of rewrite steps. To check whether or not a student follows a strategy means an ITS should parse the sequence of rewrite steps and check that the sequence of steps is a prefix of a correct sentence from the strategy as context free grammar.

The strategy language can be used for any domain. The language can be used to automatically calculate feedback on the level of strategies, given an exercise, the strategy for solving the exercise, and the student's input. Furthermore, the specification of the strategy and the calculation of feedback is separated. As a result, a strategy specification can be used to calculate different kinds of feedback.

To calculate feedback automatically, information is needed about the domain (for example the domain of logical expressions), the rewrite rules for manipulation expressions in this domain, the strategy, and common bugs made by students.

Several kinds of feedback can be given using the strategy language. Some examples are:

- Feedback can be given after each step whether or not the step is valid according to the strategy.

- Given an exercise and a strategy for solving the exercise, the minimum number of steps necessary to solve the exercise can be determined. This information can be used for showing the progress of solving the exercise.

- A student can ask for a hint. Given an exercise, a strategy for solving the exercise, and a submitted expression by the student, the best next step can be calculated.

Views are used to describe and calculate *canonical* forms [60]. For example, rewriting $1-\frac{4x+2}{3} = 3x-\frac{5x-1}{4}$ to $12-4(4x+2) = 36x-3(5x-1)$ consists of around 15 basic rewrite steps. Expanding all these steps in a derivation would made this derivation lengthy. A canonical form of an expression is a standard way of (re-)presenting that expression. Other examples are $a - b$ in stead of $a + (-b)$ and 0 instead of -0. Using views, rewrite rules can be described using a limited set of rules, only intuitive representations of expressions can be shown, rewrite steps can be described of different granularity to mimic the typical steps students take, and strategies can be recognized even when a student performs a rewrite step including some basic rewrite steps.

Depending on the maturity of a student, he or she skips one or more basic steps in a derivation. On the other hand, lectures sometimes want

to enforce some level of basic steps. Generally, students and lectures often have many wishes about customizing an ITS. The domain reasoners within the IDEAS framework are adaptable in a number of ways [61]. Examples are the granularity of the steps, which number system should be used, and the way a particular (set of) exercise(s) is solved (i.e. which strategy should be used). Examples of the last category are to remove a specific part of a strategy, to collapse a sub-strategy into a rule, to hide a rule (i.e. make the rule implicit), and to mark a sub-strategy as must-use. Furthermore, strategies can be combined into new strategies [62]. For example, two strategies can be combined using a choice combinator, a sequence combinator, and a interleaving combinator. The interleave combinator takes two strategies as argument, and allows a student to take steps from either of the two strategies, and finishes whenever both strategies are finished.

The main component of the IDEAS framework is the *domain reasoner*. The domain reasoner generates hints, analyzes rewrite steps, shows worked-out examples to an exercise, reports on common errors, decomposed an exercise into sub-exercises, etcetera. There are a number of domain reasoners as for example for solving quadratic and linear equations, rewriting logical expressions to disjunctive normal form, and to simplify and evaluate fractions, expressions using powers and square roots. Furthermore, there is a domain reasoner that supports developing small functional programs [50].

The main inputs for a domain reasoner are a domain description, the rules for manipulating the domain, strategies for solving problems, functions for testing on equality, predicates defining when a problem is solved, views and canonical forms, and functions, as for example pretty print functions, for presenting internal results to the user. The strategy language and the framework for feedback services are used in ITSs such as MathDox [32], the Digital Mathematics Environment of the Freudenthal Institute [41], and the ActiveMath system [93].

3.9 Conclusion

We have introduced a framework for providing feedback in an ITS for a well-defined domain and a well-defined task. Using the structure in the domain, we can provide detailed feedback. The framework consists of a domain with a certain semantics, a set of rewrite rules for the domain, a goal that can be reached by applying the rewrite rules in a certain order, and finally a set of indicators to determine the distance between the desired solution and the current situation. We have used our framework in a prototype ITS for solving a system of linear equations. The framework is also used in a prototype ITS

for manipulating logical formulae [71, 87].

We think the framework will be useful for students, teachers, and ITS developers that build interactive tools in which students have to construct solutions stepwise. It forces a teacher to be explicit about all terms and the semantics of a particular domain, the goal that has to be reached, how progression of the solving process can be measured, and which rewrite rules may be used. It helps the (software) developer to build the feedback component in a structural way. The steps a student can take in such a tool are not limited by a predefined set of rewrite rules provided by the tool, but can be any combination of correct or erroneous steps. The tool tries to recover the rewrite steps taken by the student in order to provide detailed feedback about possible errors. If the tool cannot recover the rewrite rules, the indicators can still help a student in determining whether or not he or she is on the right track. This is important because it is hard, if not impossible, to always determine which sequence of rewrite rules a student has applied.

An advantage of our approach is that feedback is produced for a class of problems. Furthermore, our ideas are reusable for many different classes of problems. In case of the Equation Solver feedback is given for exercises in the class of solving linear equations; instead of dedicated feedback for each separate exercise, a mechanism is defined that produces feedback for all exercises belonging to the class of solving linear equations. Most of the feedback in electronic learning environments is hard coded and specified separately for each exercise, for an example see the work by Bokhove et al. [19] on providing feedback for exercises.

We have said little about the form and content of the feedback messages. We have shown two messages: one error and one advice message. The error message not only indicates that an error has been made, but also gives the equation that contains the error, and additional information based on the rewrite analysis. This additional information is important because information about the nature of the error and the way it can be corrected is much more effective for learning than simply being informed that an error has been made without any further guidance [92]. This is especially important when students are working in a pen-and-paper like environment, instead of an environment where rewrite rules can be selected from a menu.

Our Equation Solver satisfies most of Beeson's [16] eight criteria that must be met if we are to provide successful computer support for education in algebra, trig, and calculus. The first two are *cognitive fidelity* (the software solves the problem in the same way as the student should solve it), and the *glass box principle* (a student can see how the computer solves the problem). To produce feedback and advice about which step to take next, the Equation

Solver uses a well-known set of rewrite rules and a solving strategy. The feedback and advice are based on the application of a single rule. Applying multiple rules in a single rewrite step is not supported yet, in the sense that it is allowed, but no feedback is given if an error is made, other than the fact that an error has been made. It follows that our Equation Solver does not completely meet the *customized to the level of the user* criterion. Indicators still help advanced users that apply multiple rules in one step though. The fourth criterion is the *correctness principle* (the system prevents a student from performing an incorrect operation). The Equation Solver calculates after each submission the solution of the system of linear equations. If the solution is changed, the analyzer will inform the student and, if possible, point out the erroneously applied rule. However, the student can still enter an incorrect equation. We think this has an added advantage: the student becomes aware of the syntax of systems of equations, and learns how to apply rewrite rules. The fifth criterion is *user in control* (the student decides what steps should be taken and the computer can help a student when he or she is stuck). As mentioned in the introduction, we try to stay as close as possible to the pen-and-paper situation. Instead of selecting rewrite steps from a menu, a student rewrites equations in a text field. When a student is stuck he or she presses the help button and a next step is produced. This also shows that *the computer can take over if the user is lost* criterion is satisfied. We think our Equation Solver is *easy to use*, no unnecessary typing is required, an infinite Undo is provided, and no unnecessary distractions have been added to the Equation Solver. Finally, the Equation Solver goes beyond the answer-only approach and is thus *usable with a standard curriculum*: it supports a standard curriculum in mathematics, which emphasizes step-by-step solutions.

We have implemented the Equation Solver in Haskell using standard tools from compiler technology, such as parser combinators, and pretty-printing combinators, and using the standard compiler architecture, consisting of a parsing phase, an analysis phase, and a code-generation phase. In our case the code-generation phase is the feedback-generation phase.

Chapter 4

Feedback in authoring tools

4.1 Introduction

E-learning systems, comprising ITSs, are often complex tools. Since instances of such systems, for example for a particular course, are often written by non computer science experts, authoring tools have been developed to support the development of such courses. More open-ended authoring tools for e-learning systems allow for more flexibility in both the form of the content and the order of the steps to design an e-learning system [116]. This flexibility implies a higher probability of mistakes such as inconsistencies and inaccuracies. To improve the quality of e-learning systems, an authoring tool should include mechanisms for checking the authored information on for example accuracy and consistency. Murray [116] mentions several such mechanisms. In this chapter we introduce schema analysis, with which we are able to detect a number of the possible mistakes that can be introduced by an author while authoring a course.

During authoring, different aspects of a course as for example declarative knowledge, rules, and strategies will be authored. In this chapter, we focus on *course structure* and *declarative knowledge* in the form of *domain ontologies*. Authoring these aspects falls inside the category of ill-defined tasks. Much domain knowledge is ill-defined and there is no procedure for modeling such types of knowledge. Mechanisms that check, as far as possible, the authored information for example for consistency and completeness can then

This chapter is based on: Passier, H. and Jeuring, J. (2005). Using Schema Analysis for Feedback in Authoring Tools for Learning Environments (extended version). In A. Cristea, R. Carro, and F. Garzotto, editors, Proceedings of the Third International Workshop on Authoring of Adaptive and Adaptable Educational Hypermedia, A3EH 2005, pages 13–20. [125]

be useful to support the author and function as a type of *constraints* on the knowledge structures developed [116]. Schema analysis is introduced as a technique to support the author in defining course structure and declarative knowledge. Using schema analysis, the course structure and domain ontologies can be checked on several properties. In this chapter, we represent the course structure and domain ontology by IMS Learning Design (IMS LD) [35] and Resource Description Framework (RDF) [154]. These languages will be briefly explained in this chapter.

Using IMS LD an author defines the structure of a course in a flexible way. IMS LD supports a wide range of pedagogies in e-learning. Rather than attempting to capture the specifics of many different pedagogies, it does this by providing a generic and flexible language. With such a flexible language, an author can easily make mistakes, for example an author can accidentally define an incomplete course structure. These mistakes can be partly prevented by using templates. Some drawbacks of templates are: loss of flexibility, because an author must follow the steps prescribed by a template, and problems with maintainability: it is hard to maintain documents produced by means of templates [116]. With schema analysis we maintain flexibility, are able to produce feedback when an author makes a mistake, and leave the author, as a didactic professional, free to accept or not accept the feedback information [9, 116]. The freedom to accept or not accept feedback is important. When a (possible) mistake is signaled, it is the author's decision to reject or accept the warning. Sometimes, it could be the author's intention to deviate from rules. What the system signals as a possible mistake may be correct from the author's point of view.

To determine the quality of a course, we want to detect whether or not the following properties hold for a course. If such a property holds, this may signal the absence of a potential mistake:

- *Completeness* – Are all concepts that are used in the course defined somewhere? Ideally, every concept is introduced somewhere in the course, unless stated otherwise already at the start of the course.

- *Timely* – Are all concepts used in a course defined on time? A concept can be used before its definition. This might not be an error if the author uses a bottom-up approach or inductive learning as an educational strategy rather than a top-down approach or deductive learning, but issuing a warning is probably helpful. Furthermore, if there is a large distance (measured for example in number of pages, characters or concepts) between the use of a concept and its definition, this is probably an error.

- *Recursive concepts* – Are there concepts defined in terms of themselves? Recursive concepts are often undesirable. If a concept is recursive, there should be a base case that is not recursive.

- *Correctness* – Does the definition of a concept used in the course correspond to the definition of the concept in the domain ontology?

- *Synonyms* – Are there concepts with different names but exactly the same definition?

- *Homonyms* – Are there concepts with multiple, different definitions?

Since a course and course related material are represented by means of schema languages such as IMS LD and RDF, we can use schema analysis techniques to answer the above questions, and to produce feedback about possible mistakes for authors. We have implemented the mentioned analyses as six distinct schema-analyses, which we show at work in a simple course structure and domain ontology.

Schema analysis techniques are based, amongst others, on mathematical results about fixed points [39]. Since these results are not widely known, we will explicitly show how to use them in the context of schema analysis. We will use Haskell, since this allows us to stay close to the mathematical results we use.

4.2 Schemata and schema representations

Composite objects and schemata. An ontology specifies the objects in a domain of interest together with their characteristics in terms of attributes, roles and relations. A composite object contains objects related to other objects using 'has-part' or 'uses' relations. Any object that consists of parts is called a composite object. A composite object has structure: the parts and their relations. Such a structure description is called a *schema*[1] [131]. In this chapter we focus on schemata.

Domain ontology. To represent a domain ontology we use RDF, which can be used to represent meta-data as well as the semantics of information in a machine accessible way. RDF is a universal language that describes resources. The basic building block of RDF is a triple: <resource, property, value>,

[1] Not to be confused with the concept *schema* in chapter 2.

```
<rdf: Description rdf:ID="c1">
  <do:ConceptName>Channel_capacity</do:ConceptName>
  <do:ConceptDefinition>The channel capacity is ....
                                       </do:ConceptDefinition>
  <do:uses>
    <rdf:Bag>
      <rdf:li resource="#c2"/>
      <rdf:li resource="#c3"/>
      <rdf:li resource="#c4"/>
     </rdf:Bag>
  </do:uses>
</rdf:Description>

<rdf:Description rdf:ID="c2">
  <do:ConceptName>Bandwidth</do:ConceptName>
  <do:ConceptDefinition>The Bandwidth is ....
                                       </do:ConceptDefinition>
</rdf:Description>
```

Figure 4.1: Domain ontology represented in RDF

which defines concepts and related concepts. For example the concept 'cycle-wheel' consists of (has parts) the concepts 'rim' and 'spoke', i.e. <cycle-wheel, has-part, rim> and <cycle-wheel, has-part, spoke>.

We show as an example the concept 'channel capacity' from the domain communication technology. It defines how many bits ('0' or '1') per second can be transmitted. The channel capacity depends on (uses) three other concepts: 'bandwidth', 'signal power' and 'noise power'. A possible (incomplete) schema for channel capacity is shown in figure 4.1. The structure of the concepts 'signal power' and 'noise power' is the same as for the concept 'bandwidth'. The alias 'do' is the abbreviation of domain ontology; an own specified namespace.

Most of the code in figure 4.1 is self-explanatory. The Bag-tag is used to model a list in which order is not important. `#c2` within the li-tag means a reference to resource c2.

Course structure. XML is a language for structuring documents. A data type definition (DTD) describes the type of a set of XML documents. IMS LD is a DTD developed to represent structures of electronic courses. The content of a course is presented in a structured way, and activities in an activity-structure. For the examples in this paper we focus on the Activity-model,

```
1.  <!ELEMENT Activity %Activity-model;>
2.  <!ATTLIST Activity
3.            ...
4.            Educational-strategy (Inductive | Deductive)>
5.  <!ENTITY %Activity-model "(Meta-data?,
6.                                ...,
7.                                Activity-description)">
8.  <!ELEMENT Activity-description (Introduction?, What, How?,
9.                                    ..., Feedback-description?)>
10. <!ELEMENT What %Extra-p; >
11. <!ENTITY %Extra-p "(... | Figure | Audio | Emphasis | List |
12.                     ... | Example| Definition)*">
```

Figure 4.2: Parts of the activity-model in IMS LD definition

which consists of several elements: Meta-data, Objectives, Prerequisites, Environment and an Activity-description. Figure 4.2 shows an example. An activity-description consists of nine elements. One of them is the What-element, which contains the instruction for the activity to be performed. Possible instructions are grouped together by the parameter entity Extra-p. To be able to add more specific annotations to content and structure we introduce two new elements in the Extra-p element, namely Definition and Example (see line 12 in figure 4.2). Furthermore, we introduce a new attribute Educational-strategy of the element Activity with two possible values: Inductive and Deductive (see line 4 in figure 4.2). Introducing such elements will make it possible to structurally analyze educational material. These elements serve as examples to illustrate the analysis techniques at work. In practice many elements can be added, depending on the desired analyses. Figure 4.2 shows only the relevant elements and attributes related to the activity-model together with the newly defined elements example and definition. The definitions of the new elements Definition and Example are presented in figure 4.3.

4.3 Schema analysis to detect authoring problems

The schemata given in section 4.1 represent structural aspects, which can be analyzed. In this section we give some examples of schema-analyses that determine whether or not certain properties hold. The results of these analyses form the basis of feedback to the author. The analyses take the schemata as input.

```
<!ELEMENT Definition (Description, Concept, RelatedConcept+)>
<!ATTLIST Definition Id ID #REQUIRED
        Name CDATA #REQUIRED>

<!ELEMENT Example (Description, Concept, RelatedConcept+)>
<!ATTLIST Example Id ID #REQUIRED
      Name CDATA #REQUIRED
      Belongs-to-definition IDREFS #REQUIRED>

<!ELEMENT Description (CDATA)>

<!ELEMENT Concept EMPTY>
<!ATTLIST Concept Id ID #REQUIRED
          Name CDATA #REQUIRED
<!ELEMENT RelatedConcept EMPTY>
```

Figure 4.3: Definition of the new elements

In this chapter we perform two types of analyses:

1. the analysis of structural properties of a schema, for example the recursive property, and

2. the comparison of a schema with one or more other schemata, for example to test the correctness of a definition.

Since a schema is very similar to a context-free grammar we can use grammar analysis techniques [72] to analyze a schema for these structural properties. For example, to find out whether or not a concept is recursive, we have to calculate, for each concept, the concepts it depends on. How do we calculate these dependencies for an arbitrary ontology? We describe an answer to this question, together with examples of schema analysis, in the following sections.

4.3.1 Data structures and definitions

We represent an (domain) *ontology* with a compositional view as a list of concept definitions. A *concept definition* is a tuple consisting of a concept identifier `Id` and a bag of related concepts, `RelatedConcepts`, in which the concept identifiers may appear in any order. In this definition we abstract for instance from concept name, attributes and cardinalities.

```
data Ontology         =  Ont [ConceptDef]
```

```
type ConceptDef       =  (Id, RelatedConcepts)
type RelatedConcepts  =  Bag
type Bag              =  [Id]
type Id               =  String
```

Note that a data type definition in Haskell introduces a constructor function for the data (Ont in the case of Ontology), whereas type definitions use the constructors of the types used.

The structure of the data type Course follows the IMS LD definition (see figure 4.2) and consists of an identifier and a list of activities. Extra-p is limited to example and definition.

```
data Course                = C(Id,[Activity])
type Activity              = (Id,EducationalStrategy,[Extra_p])
data EducationalStrategy = Inductive | Deductive
data Extra_p               = Ex ( Id
                                 , ConceptId
                                 , RelatedConcepts
                                 , DefRefs
                                 )
                           | Def ( Id
                                  , ConceptId
                                  , RelatedConcepts
                                  )
type RelatedConcepts       = Bag
type DefRefs               = Bag
type ConceptId             = Id
```

`terminalConcepts` are the set of concepts with no related concepts, `nonTerminalConcepts` the set of concepts with at least one related concept, and `allConcepts` the set of all concepts. Function reachable

```
reachable::[ConceptDef] -> [ConceptDef] -> [ConceptDef]
```

takes `nonTerminalConcepts` and `allConcepts` as input and returns for each concept the set of all concepts that are reachable, both directly and indirectly. Function `reachableTerminals`

```
reachableTerminals::[ConceptDef] -> [ConceptDef] -> [ConceptDef]
```

takes `nonTerminalConcepts` two times as argument and returns for each concept the terminal concepts that are reachable. Suppose for example the following ontology:

```
Ont [(a,[b,c]),(b,[]),(c,[d,e]),(d,[]),(e,[])]::Ontology
```

then:

- `terminalConcepts` is: `[(b,[]),(d,[]),(e,[])]`
- `nonTerminalConcepts` is: `[(a,[b,c]),(c,[d,e])]`
- `allConcepts` is: `[(a,[b,c]),(b,[]),(c,[d,e]),(d,[]),(e,[])]`
- `reachable nonTerminalConcepts allConcepts` = `[(a,[b,c,d,e]),(b,[]),(c,[d,e]),(d,[]),(e,[])]`
- `reachableTerminals nonTerminalConcepts nonTerminalConcepts` = `[(a,[b,d,e]),(c,[d,e])]`.

Both functions, `reachable` and `reachableTerminals`, use a fixpoint calculation implemented by function `limitBy`. Function reachable is defined by:

```
reachable productions conceptdefinitions =
 limitBy equalConceptDefs (expand productions) conceptDefinitions
```

Function `expand` expands the concept definitions using a set of productions: if production (β, ω) is used, all related concepts `xs`++$[\beta]$++`ys` are expanded to `xs`++$[\beta]$++ω++`ys` removing duplicates. Function `limitBy` repeatedly applies function `expand` until a fixpoint is reached, after which `conceptDefinitions` contains for every concept all reachable concepts. Function `equalConceptDefs` determines the fixpoint. A fixpoint is reached if two successive `conceptDefinitions` are equal.

Suppose for example ontology `o` and function call

```
reachable productions conceptDefinitions
```

where

```
productions = [(a,[b,c]),(c,[d,e])]}
conceptDefinitions = [(a,[b,c]),(b,[]),(c,[d,e]),(d,[]),(e,[])].
```

After the first iteration `conceptDefinitions` equals `[(a,[b,c,d,e]),(b,[]),(c,[d,e]),(d,[]),(e,[])]`. After the second iteration `conceptDefinitions` is unchanged, what means a fixpoint is reached, and contains for each concept the set of all reachable concepts.

Function limitBy is defined as:

```
limitBy::(a -> a -> Bool) -> (a -> a) -> a -> a
limitBy eq h s | eq s next  = s
               | otherwise  = limitBy eq h next
               where next = h s
```

Function `limitBy` only terminates if its argument function `(a -> a)` is continuous on a complete partial order or CPO [39], which informally means that the argument function should be increasing on a restricted domain.

The determination of `reachableTerminals` is calculated in a similar way. Instead of function `expand` function `derivationStep` is used. With production (β, ω), all related concepts `xs++`β`++ys` are changed to `xs++`ω`++ys`. More details about efficient algorithms can be found in [72].

4.3.2 Solving authoring problems with schema analysis

In this section we describe solutions to schema analysis problems, which can detect the (possible) mistakes listed in the introduction of this chapter.

Completeness. We distinguish three kinds of (in)completeness:

- within a course,
- within a domain ontology, and
- between a course and a domain ontology.

If a concept is used in a course, for example in a question or an example, it has to be defined elsewhere in the course. To determine this for all concepts, we define the function `completeCourse`:

```
completeCourse :: Course -> Bool
completeCourse  =  null.undefinedConceptsCourse
```

The input parameter of `completeCourse` is of type `Course`, the output is a boolean. Function `completeCourse` uses function `undefinedConceptsCourse`, which lists the undefined concepts in a course. Function `completeCourse` tests (by means of function `null`) the emptiness of this list: if the list is empty, then the course is complete (`null [] = True`).

Determining the undefined concepts in a course is calculated in three steps:

1. Take the set of all concepts that appear in the right- and left hand sides of concept definitions within all examples and all concepts that appear in the right hand side of concept definitions within all definitions (`usedConcepts`).
2. Take the concepts that appear in the left-hand site of the concept definitions (`definedConcepts`).
3. check that each of the used concepts appears in the set of defined concepts (function `diffBag`).

If all concepts used appear in the set of defined concepts the result of `undefined- ConceptsCourse` is the empty set, otherwise the list of concepts without a definition.

```
undefinedConceptsCourse c =
  let usedConcepts    = extractUsedConceptsCourse c
      definedConcepts = extractDefConceptsCourse c
  in  diffBag usedConcepts definedConcepts
```

Functions for determining the completeness property can also be applied to an (domain) ontology (`completeOntology`, which uses an `Ontology` as argument), and between a course and a domain ontology (`completeCours-Ontology`, which uses a `Course` and an `Ontology` as arguments). Function `completeOntology` checks if all used concepts in the ontology are defined in the same ontology. Function `completeCourseOntology` checks if all used concepts in a course are defined in the ontology. The same three steps are performed in both functions.

Timely. A concept can be used before it is defined. This might not be an error if the author uses an inductive instead of a deductive strategy to teaching, but issuing a warning is probably helpful. Furthermore, there may be a large distance (measured for example in number of pages, characters or concepts) between the definition and the use of the concept, which is probably an error. We define function `timely` to determine whether or not concepts in a course are defined in time and a function `outOfOrderConcepts` to list the concepts that are out of order.

```
timely :: Course -> Bool
timely = null.outOfOrderConcepts
```

In function `outOfOrderConcepts`, function `extractActivities` returns a list `activities` with for each each activity in the course the tuple `(Estrategy,`

[Extra_p]). Then, using functions inits and tails every [Extra_p] list is split as follows: for every element x in the list [Extra_p] the list is subdivided into a left part (epl), which contains all elements to the left of element x, and a right part (epr), which contains element x as and all elements to the right of x. For example, for the input list [e,d] we get [([],[e,d]),([e],[d]),([e,d], [])], where e is example and d is definition. Finally, function intime tests the timely constrains for all tuples (es,(epl,epr)): if the first element of epr is a definition and the educational strategy is deductive, then: (1) a related example appears after the definition, and (2) no related example appears before the definition (tested by elemBy eqConcept c in the code below). In case of an inductive activity, a related example appears before the definition and no related example appears after the definition. Function intime is always true if epr is empty or the first element of epr is an example.

```
outOfOrderConcepts :: Course -> [Extra_p]
outOfOrderConcepts c =
  let activities = extractActivities c
      split      = [ (es,s) | (es,eps) <- activities
                            ,  s <- zip (inits eps) (tails eps)
                   ]
  in [head epr | (es,(epl ,epr)) <- split
               , not (intime (es,epl,epr))
     ]

intime (_,_,[])                              = True
intime (_,_,Ex (j,c,cs,r):_)                 = True
intime (Deductive, epl, Def (j,c,cs):epr) =
  elemBy eqConcept c epr && not (elemBy eqConcept c epl)
intime (Inductive, epl, Def (j,c,cs):epr) =
  elemBy eqConcept c epl && not (elemBy eqConcept c epr)

eqConcept id (Def (i,c,cs))  = False
eqConcept id (Ex (i,c,cs,r)) = id == c
```

Recursive concepts. A concept can be defined in terms of itself. Recursive concepts are often not desirable. If a concept is recursive, there should be a base case that is not recursive. Recursive concepts may occur in a course as well as in an ontology. We define two functions: recursiveOntology and recursiveCourse which take an ontology respectively a course as argument.

Both first extract all concept definitions, and use function `recursiveConcepts`. We show the definition of recursiveOntology.

```
recursiveOntology :: Ontology -> Bool
recursiveOntology = not.null.listRecursiveConceptsOntology

listRecursiveConceptsOntology :: Ontology -> [Id]
listRecursiveConceptsOntology =
  recursiveConcepts.extractAllConceptsOnt
```

Function `recursiveConcepts` calculates for every concept all reachable concepts, as explained in section 4.3.1. Every concept in `reachables` is checked for recursiveness: a concept is recursive if the concept's Id is a member of the set of the reachable concepts. The recursive concepts are collected in a list.

```
recursiveConcepts :: [(Id, RelatedConcepts)] -> [Id]
recursiveConcepts allConcepts =
  let nonTerminalConcepts  = filter (not.null.snd) allConcepts
      reachables           =
        reachable nonTerminalConcepts allConcepts
  in [x |(x, y) <- reachables, elem x y]
```

Correctness. Concept definitions in a course should correspond with the same concept definitions in the domain ontology. To solve this problem, for every concept in a course all reachable terminal concepts are determined by function `reachableterminals`. This characterisation is compared against the reachable terminal concepts based on the domain ontology using function `verifyCorrectness`. The definition of `listIncorrectConcepts` is:

```
listIncorrectConcepts c o =
  let allConceptsOnt           = extractAllConceptDefsOnt o
      nonTerminalConceptsOnt   =
        filter (not.null.snd) allConceptsOnt
      reachableTerminalsOnt    =
        reachableTerminals nonTerminalConceptsOnt allConceptsOnt
      allConceptsCourse        = extractAllConceptsDefCourse c
      reachableTerminalsCourse =
        reachableTerminals nonTerminalConceptsOnt allConceptsCourse
  in verifyCorrectness reachableTerminalsCourse
                       reachableTerminalsOnt
```

and function correct calls `listIncorrectConcepts`.

```
correct :: Course -> Ontology -> Bool
correct c o  = null (listIncorrectConcepts c o)
```

Synonyms. Concepts with different names may have exactly the same definition. For example, concept a with concept definition `(a,[c,d])` and concept b with concept definition `(b,[c,d])` are synonyms. Per definition, as an example, we declare two concepts `x` and `y` as synonyms if their identifiers are different and `(reachableTerminals nonTerminalConcepts x)` equals `(reachableTerminals nonTerminalConcepts y)`. We define function synonyms to check for synonyms in an ontology, which uses function `listSynonyms` to lists the synonyms. In `listSynonyms`, first `nonTerminalConcepts` is determined from all concept definitions. Secondly, for all concepts in the ontology all reachable terminal concepts are determined by function `reachableTerminals`. In a last step the concept definitions with the same right hand side and different left hand sides, using function `equalRhs`, are collected in a list. The formal definition of `listSynonyms` is:

```
synonyms :: Ontology -> Bool
synonyms = not.null.listSynonyms

listSynonyms :: Ontology -> [([Id],RelatedConcepts)]
listSynonyms o    =
  let allConcepts                = extractAllConceptDefsOnt o
      nonTerminalConcepts        = filter (not.null.snd) allConcepts
      reachableTerminalsConcepts =
        reachableTerminals nonTerminalConcepts nonTerminalConcepts
      equalrhs                   = equalRhs reachableTerminalsConcepts
  in [c | c <- equalrhs, length (fst c) > 1]
```

Homonyms. A concept may have multiple, different definitions. Suppose for example concept `a` with concept definitions `(a,[b,c])` and `(a,[d,f])`. In this case there is a matter of homonym. To list the homonyms in a domain ontology, we define:

```
listHomonyms = kpDups.extractDefConceptsOnt
```

which has an ontology as argument. Function `extractDefConceptsOnt` extracts all left hand sides of the concept definitions and returns a list of concept identifiers `[Id]`. Function `kpDups` takes this list as input and returns

a list with duplicated identifiers. The corresponding function homonyms is defined as:

```
homonyms :: Ontology -> Bool
homonyms = not.null.listHomonyms
```

4.4 Related work

Although many authors underline the necessity of feedback in authoring systems [7, 9, 116], we have found little literature about feedback and feedback generation in authoring systems.

Jin et al. [74] describe an authoring system that uses a domain as well as a task ontology to produce feedback to an author. The ontologies are enriched with axioms, and on the basis of the axioms the models developed can be verified and messages of various kinds can be generated when authors violate certain specified constraints. The details of the techniques used are not given, and it is not clear to us how general the techniques are. Our contribution is the introduction of schema analysis as a general technique to produce messages about errors of structural aspects of course material.

Aroyo et al. [7, 8, 9] describe a common authoring framework. The framework contains a domain as well as a task ontology and supports an authoring process in terms of goals, and primitive and composite tasks. Based on ontologies, the framework monitors and assesses the authoring process, and prevents and solves inconsistencies and conflicting situations. Their requirements for authoring support are:

1. help in consistently building courseware,
2. discovery of inconsistencies and conflicting situations,
3. production of feedback, hints and recommendations, and
4. modularisation of authoring systems (reusability).

We think that our framework satisfies all these requirements. Schema analysis as a technique could be positioned in 1, 2 and 4.

Stojanovic et al. [142] present an approach for implementing e-learning scenarios using the semantic web technologies XML and RDF, and make use of ontology based descriptions of content, context and structure. A high risk is observed that two authors express the same topic in different ways (homonyms). This problem is solved by integrating a domain lexicon in the ontology and defining mappings, expressed by the author itself, from terms

of the domain vocabulary to their meaning defined by the ontology. In our approach these mappings are analyzed automatically.

In the Authoring Adaptive Hypermedia community the importance of feedback mechanisms in authoring systems has been recognized [38]. Although we have found an impressive amount of authoring tools for adaptive hypermedia [26], we have not found descriptions of technologies used for providing feedback to authors. We expect our results will be useful for authoring adaptive hypermedia as well.

4.5 Conclusion

In this chapter, we described our framework towards the production of feedback based on schema analysis in an e-learning environment and treated some analyses in detail. The framework supports the general requirements described in the literature: reusability, flexibility and the production of semantically rich feedback. Our approach make use of modern XML-based languages as RDF and IMS LD. Six analyses on structural aspects of e-course related material are described and specified using the functional language Haskell.

Chapter 5

Supporting several model languages

5.1 Introduction

As is described in chapter 1, models are represented using special languages. There are many modeling languages. Examples are the Unified Modeling Language (UML) [85], a language specifically for object oriented modeling, the Entity-Relationship technique (ER) [30], and the Object Constraint Language OCL [155]. Most modeling languages are graphically oriented. These graphical languages are popular, because they are easy to use and appeal intuitively. Examples of graphical languages are UML and ER. An example of a textual language is OCL. In this chapter we focus on UML class diagram as graphical model language.

Students in Computer science have to learn one or more modeling languages, for example one graphical and one textual language. Each language has its pros and cons and is suitable for certain situations or domains. For example, graphical languages are suitable for communicating models to people, whereas textual languages are suitable for computer analysis. Furthermore, students need to practice a lot in order to develop truthful models. In modeling, a good solution is a schema that is syntactically correct and expresses the right semantics. Examples of the syntax of a language and the semantics of a model are given in section 5.2.2 and 5.2.3.

If a student is practicing modeling, he or she can make many errors.

This chapter is based on: Passier, H. (2008). A framework for feedback in e-learning systems for data modeling. In Proceedings of the IADIS International Conference, e-Society. [119]

Globally, we can distinguish three types of errors:

1. Errors concerning the *syntax* of the language used to represent a model.

2. Errors concerning the structure of a model, or *meta-model* errors.

3. Errors concerning the *semantics* the model should express.

Examples of both types of errors are given in section 5.2.4.

As is described in chapter 2, there are some ITSs for practicing modeling. All of these systems provide feedback on the syntax of a model. To provide feedback on the semantics of a model, the ITSs make use of a set of constraints or an author's model to compare with. If feedback about semantics is presented, it is often hard coded and part of the exercise. The development and implementation of this type of exercises is labor intensive and requires special knowledge and skills. The situation worsens in case students have to practice several modeling languages. This implies that for every model language a new set of exercises has to be implemented, where the feedback part of each exercise must be implemented separately. To overcome this problem, we present the outline of a framework that:

1. generates, besides feedback about syntactic errors of the language used, feedback about the semantics of a model;

2. is able to generate feedback for several model languages.

In section 5.3 the ideas of this framework are presented and demonstrated using a small example. In this chapter, we restrict ourselves to structural models, models that represent the structure of the system or the data the system processes. Examples are UML class diagrams, ER diagrams, and CC diagrams. Simple UML class diagrams consisting of classes and associations are used as example.

5.2 UML-Class diagrams

5.2.1 An example

The process of constructing a model is illustrated using a simple example: a limited UML-class diagram consisting of classes, associations and multiplicity constraints. In educational contexts, a student is usually given a description like the following:

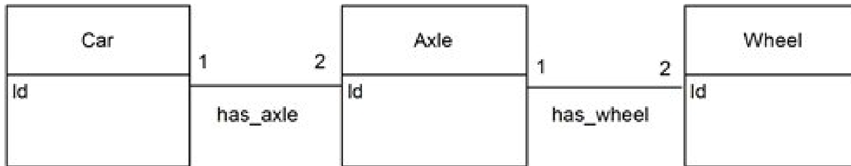


Figure 5.1: A UML class diagram as student's solution

'You should construct a model for an software application that processes and stores data about cars in a car factory. A car consists of axles and wheels. The database should store the axles connected to each car and the wheels connected to each of the axles.' Etcetera.

The student's task is to construct a model from this description. It is obvious that car, axle and wheel are of importance, and the student may decide to represent these concepts as classes. As can be seen in figure 5.1, the student has assigned an attribute Id to each of the classes, which is not mentioned in the description. The student also needs to identify the associations between these three classes and the corresponding multiplicities. Information about this is not given in the text, so the student has to make an own decision. The student has decided to draw the associations 'has axle' and 'has wheel' with multiplicities as shown. The association between the classes car and axle should be read as: every car has two axles and every axle is connected with one car.

There are many things a student has to know and think about when constructing a model. He or she has to understand the domain of interest, the model language, the use of integrity constraints, etcetera. In real educational situations, the text is mostly longer and often contains some ambiguities. In a nutshell, modeling is not a well-defined process and the task is open ended: there is no single best solution for a problem, and there are often several correct solutions in relation to the same set of requirements.

5.2.2 Syntax

The syntax of a model language describes the possible configurations of the model elements and how they are represented.

Classes. Classes categorize the objects that can exist in a system and define their shared properties. Class diagrams show the classes in a system and a variety of relationships between those classes. The basic notation of

classes could consist of three compartments. The top compartment, which is obligatory, contains the class name. The second compartment is optional and contains the attributes of the class. The third compartment contains the operations of the class. This compartment is optional too and is not used in the example.

Associations. Classes can be connected by means of associations. An association between two classes implies a link between instances of the classes. Associations can optionally be labeled with a name together with an arrowhead indicating the direction in which the name should be understood. Association-ends can optionally be labeled with multiplicities.

Multiplicities. Multiplicities are used to specify how many occurrences of a model element are permitted in a context. Generally, a multiplicity specification consists of an interval definition of the form *lowerbound..upperbound*. The lower bound can be any non-negative integer, the upper bound can be any non-negative integer or the symbol '$\star$', which denotes that the range is unbounded. If the lower bound equals the upper bound, the range specifies a single number and can be written as such.

5.2.3 Semantics

The semantics of a model imply the facts in the world the model refers to. Without semantics, a model is just an arrangement of graphical elements on a page. With semantics, each part of a model makes a claim about the world. In case of figure 5.1, the semantics of the model say that there exist three categories of objects: cars, axles and wheels and that every car has exactly two axles and every axle has exactly two wheels. In terms of databases, the semantics of a model say which information can be extracted from the database using a query language. Following figure 5.1, we could ask the car factory database which wheels are connected to which car for example.

5.2.4 Types of errors

We can distinguish three types of errors:

- Syntactic errors
- Meta-model errors
- Semantic errors

Syntactic errors. Syntactic errors occur when a student violates a syntactic rule. In case of class diagrams, examples are two or more classes having the same name, or an association with a dangling side.

Meta-model errors. Examples of meta-model errors are inconsistency and redundancy in a model.

Consistency is a property that must hold between related models and within a model. An example of the first category is that if a sequence diagram uses an object, the class of the object should be defined in the related class diagram, otherwise there is an inconsistency between both diagrams. An example of the second category is that every diagram should guarantee a finite but not empty reality, i.e. there exist at least one instance of the diagram.

Figure 5.2 shows two inconsistent UML class diagrams. The first one (a) is inconsistent, because the model assumes an infinite binary tree of objects of type A[1]. In the second one (b), showing an example for a library system, there are two classes Member and Book and an association which shows the relationship between members and books. *Every* member can borrow at most five books at the same time. Every teacher however, where every teacher is a subtype of Member, can borrow at least six books. As a consequence, the diagram is inconsistent.

Redundancy expresses itself by cycles in models from what associations can be possibly removed without loss of information. Whether or not an association can actually be removed, depends on the semantics of the association. It is a human who has to decide about this. The candidate removable associations can be derived and reported.

An example of redundancy is shown in figure 5.3: the cycle contains a redundant association, because each car has two axles and each axle has two wheels. So, one can derive that every car has four wheels and that every wheel is connected to one car. This is exactly what the association 'drives on' represents. One of the associations in figure 5.3 is redundant and can be removed.

Semantic errors. A semantic error occurs when the model does not determine those facts in the world the model refer to, that means the domain of interest. Examples of semantic errors are wrong facts (for example a car

[1]Notice that to enforce infinity, an OCL constraint is needed specifying that no A-object can be connected to itself.

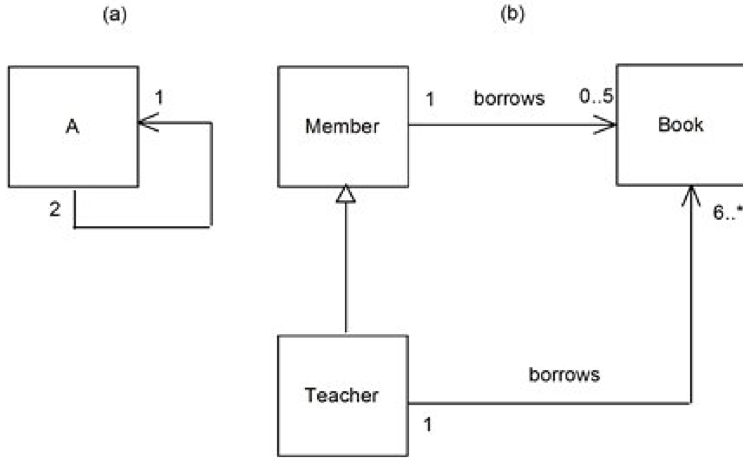


Figure 5.2: Two inconsistent models

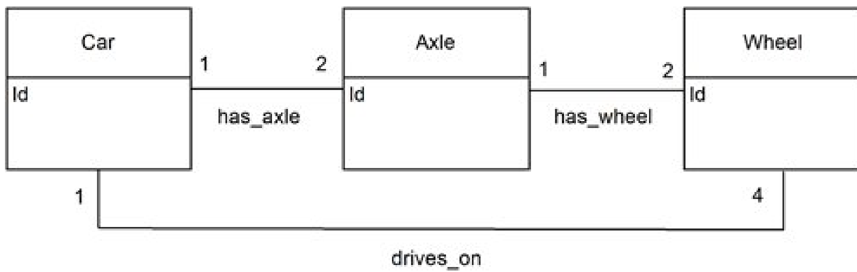


Figure 5.3: A redundant model

has one wheel instead of four wheels), loss of information (for example the class Wheels and the related association 'has wheel' are missed), and addition of undesirable information (for example a class called Factory has been added to the model).

In the remainder of this chapter, we will focus on the last two types of errors, i.e. meta-model and semantic errors. An exercise as described in section 5.2.1 is assumed: a student is given the requirements in natural language about a particular domain and he/she has to construct a class diagram fulfilling the requirements.

5.3 The framework

To generate feedback on the level of meta-model and semantics for several languages, the framework assumes we have the following main parts:

A central model. For each kind of exercise there exists a central model (or author's model) including all information needed to evaluate the student's model on the *semantics* the model should express. In cases of more complex models with several possible solutions, 'the central model' may consist of a number of solution models. Because the framework has to support several model languages, the central model is represented using a model language, mostly different from the student's model, with enough expressive power to include all models which can be represented by all languages allowed. Furthermore, to obtain automatic feedback, the language must be mathematically rigorous and processable by a computer. The concept 'central model' will be explained in section 5.3.1.

Translations. Students might practice modeling using several languages. For each model language, there exists a function that translates a student's model into the language of the central model and vice versa. Due to the differences in expressive power between model languages and the possibility that the central model consists of several solution models, each student's model can be translated to at least one central solution model and each central model can be translated into at least one model language used. The student's model translated and the central model are input for certain analyses. The concept translation will be explained in section 5.3.2.

Analysis functions. There is a set of *semantic analysis* functions. Each of these functions takes a (part of a) translated student's model and a central

model as arguments and returns the result of certain semantic analyses. Furthermore, there is a set of *meta-model analysis* functions. Each of these functions only takes a (part of a) translated student's model as argument and returns the result of certain meta-model analyses. The results of both analyses form the raw information for feedback to be presented to a student. The concept analysis function will be explained in section 5.3.3.

5.3.1 The central model

For each exercise there exists a central model. Generally, such a model includes all information needed to express the specific structures and constraints as part of the exercise. The central model is used to evaluate the student's model on semantic correctness and completeness (see section 5.3.3). The language(s) used to expresses central models must fulfill some requirements. For example, the language must:

- have enough expressive power to express all specific structures and constraints desired as part of the exercises;
- have enough expression power to include all structures and constraints which can be modeled by all languages students might practice with;
- be relative easy in use and accessible for a large group of people, i.e. lecturers and authors of the exercises;
- be rigorous enough to make precise reasoning by a computer possible.

To capture all requirements stated, we use *theory of relations* and *predicate calculus* to represent models. It is widely accepted that these languages can be considered as a complete and all encompassing framework [131, 149]. The difficulty of reading and writing predicate calculus is a disadvantage. Alternatively, predicate sentences can be represented pictorially, as is used in semantic network models [131, 149]. In this chapter, we will focus on simple class diagrams only and these structures are implemented as *concepts* and *relations* between concepts. A central model then is represented as a record consisting of these two fields (figure 5.4). Relations as well as concepts are implemented using the data type `Set`. The types `Concepts` and `Relations` will be discussed in following paragraphs.

Concepts. Informally, a concept is a physical or an abstract thing of interest. A concept is considered to be determined by its extent and its intents. The extent consists of all objects belonging to the concept, while the intent

```
data CModel = CM {concepts :: Concepts, relations :: Relations}
```

Figure 5.4: The central model

```
data Concept = Concept{ conceptname :: String
                      , category :: MLontology
                      }

type Category = MLontology
type Concepts = Set Concept

newConcept :: (ConceptName, Category) -> Concept
newConcept (n, c) = Concept {conceptname = n, category = c}
```

Figure 5.5: Representation of concept

is the collection of all attributes shared by the objects. In relation to simple UML class diagrams, concepts are the classes and attributes belonging to the classes. We represent a concept as a record with two fields: `conceptname` and `category` (see figure 5.5). The first field represents the name of a concept and functions as an identifier too. The second field categorizes the origin of the concept in relation to the model language used. Assuming simple UML class diagrams, possible values are `Individual`, `Attribute` and `Relation`. Classes are categorized to `Individual`, whereas attributes and relations are categorized to `Attribute` respectively `Relation`. We will discuss this matter further in section 5.3.2. As can be seen in figure 5.5, the type concepts is defined as a set. To create a new set of concepts, we use function `empty`.

Relations. For two concepts A and B, any subset of $A \times B$ is a relation from A to B. We represent a relation as a record with six fields: `relationname`, `leftconcept`, `leftcard`, `rightconcept`, `rightcard` and `category` (see figure 5.7). The first field represents the name of the relation. The second and fourth fields identify both concepts related. The third and fifth fields express cardinalities. Figure 5.6 shows an example: a car has exactly two axles (the minimal cardinality equals the maximal cardinality), and each axle is connected to exactly one car. Again, the field `category` categorizes the origin of the relation. In this case the value equals Relation.

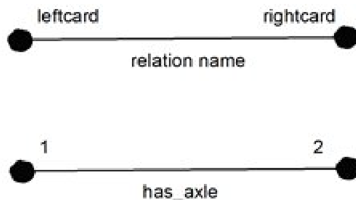


Figure 5.6: Relations

As can be seen in figure 5.7, the type `Relations` is defined as a set. Cardinality is implemented using a data type `Cardinality`, which is a record with fields minimum and maximum cardinality. Possible values are positive integers (`C Int`) or infinity (`Infinite`). An extra function is used to verify the invariant: $mincard \leq maxcard$, which is not shown here.

Student's model. In our case of simple class diagrams, the type of a student's class diagram looks like the type of the central model (figure 5.8). Again, a class-diagram is represented as a record with two fields: classes and associations. Every class has a class name which functions as an identifier too. Furthermore, a class could have a set of attributes. In this example, we will model an attribute as a string representing the name of the attribute and abstract for example from typing. Cardinalities are slightly different too. In case of UML models, we talk about multiplicity. The minimum value could be any positive integer (`M Int`), the maximum value could be any positive integer or has a unbounded range (`star`).

5.3.2 Translations

In case of UML class diagrams, a function transforms a class diagram into an instantiation of the central model and vice versa. This transformation must satisfy the property of *invertibility.* Suppose a function `cd2cm` that takes a class diagram (cd) as argument and returns the corresponding central model (cm). If there exists an inverse relation `cm2cd` in the opposite direction, with the property that a round trip `cd2cm.cm2cd` from a certain class diagram to an instantiation of the central model and back returns the same class diagram in terms of syntax and semantics, then `cd2cm` is called invertible.

To be able to satisfy the property of invertibility, we make use of *Chisholm's ontology* [100]. Chisholm's ontology is selected because this ontology tends to have elements common to all of the modeling languages, such as defining

```
data Relation = Rel { relationname  :: String
                    , leftconcept   :: String
                    , leftcard      :: Cardinality
                    , rightconcept  :: String
                    , rightcard     :: Cardinality
                    , category      :: Category
                    }

data Cardinality  = Card {mincard:: Card, maxcard:: Card}
data Card         = C Int | Infinite
type Relations    = Set Relation

newRel :: (RelName, LeftConcept, Cardinality,
           RightConcept, Cardinality, Category) -> Relation
newRel (n,lcon,lcard,rcon,rcard,cat) = Rel { relationname = n
                                           , leftconcept  = lcon
                                           , leftcard     = lcard
                                           , rightconcept = rcon
                                           , rightcard    = rcard
                                           , category     = cat
                                           }
```

Figure 5.7: Representation of relations

```
data ClassDiagram = CD { classes      :: Set Class
                       , associations :: Set Association
                       }

data Class        = Class { classname  :: String
                          , attributes :: Set String
                          }

data Association  = Assoc { assocname  :: String
                          , leftclass  :: String
                          , leftmul    :: Multiplicity
                          , rightclass :: String
                          , rightmul   :: Multiplicity
                          }

data Multiplicity = Mul {minmul :: Mult, maxmul :: Mult}
data Mult         = M Int |  Star
```

Figure 5.8: UML-Class diagram

and describing objects, individuals, entities and their relations. In our case of transforming simple class diagrams we only use the categories: Individuals, Attributes and Relations:

- *Individuals* – These are the objects of interest. Individuals come into being (are created) and past away (are destroyed), so they are transient. Each individual possesses an attribute or several attributes that uniquely identifies it. Individuals may have constituents and can have thereby structure.

- *Attributes* – Attributes are exhibited by individuals. Attributes are enduring and are coupled with individuals.

- *Relations* – Individuals may be related. Specifically, relations are attributes (an ordered pair) identifying the participating individuals required.

Figure 5.9 shows the representation of type `MLOntology` (Model Language ontology) and the categories used. Using these categories we are able to transform a class diagram into an instantiation of the central model satisfying the property of invertibility:

```
data MLontology = Individual | Attribute | Relation
```

Figure 5.9: Categories used from Chisholm's Ontology

Associations. Each association a is transformed by creating a new relation:

```
newRel( assocname a
      , leftclass a
      , mul2card (leftmul a)
      , rightclass a
      , mul2card (rightmul a)
      , Relation
      ).
```

Function `mul2card` takes a `Multiplicity` and returns the corresponding `Cardinality`. Furthermore, the field category is set to value `Relation`.

Classes. Each class can contain zero or more attributes. A class c is transformed by creating a new concept: `newConcept (classname c, Individual)`, where category is set to `Individual`.

Attributes. Each attribute a is transformed into a concept and a relation. The new concept equals `newConcept (attributename, Attribute)`, where category is set to `Attribute`. The new relation connects concept c representing the class the attribute belongs to and concept c' representing the attribute itself:

```
newRel( "attr"
      , conceptname c
      , Card { mincard = C 1, maxcard = C 1 }
      , conceptname c'
      , Card { mincard = C 1, maxcard = C 1 }
      , Attribute
      )
```

Remark that this relation is injective (a one-to-one function) and category is set to `Attribute`.

5.3.3 Analysis functions

As said in section 5.2.4 we distinguish three type of errors, namely syntactic errors, meta-model errors and semantic errors. We use the same classification for analysis functions. The main interest in this chapter is on meta-model errors and semantic errors. For both types of errors an example is given.

Syntactic errors. As said before, this type of error and analysis is not of interest in this research. Many UML-editors are able to analyze class diagrams on syntactic errors and transform the analysis results into feedback.

Meta-model errors. The analysis of meta-model errors is applicable on all models. Examples are the presence of inconsistency and redundancy. Here, we give an example of redundancy (see section 5.2.4). For searching for redundancy, we are looking for cycles where one or more relations can be removed without loss of information. To solve this process, three phases are performed. Firstly, for every concept c^* part of the UML-class diagram all paths $c^* - r - c - r - ...$ are determined, where c is a concept and r is a relation, with c^* as starting point. For this, a depth first search (DFS) algorithm is used. The search is stopped when a cycle is found, i.e. a visiting concept is already part of the list of visited concepts. If a concept is reached without any (outgoing) relation anymore and no cycle is determined the path is removed. In a second phase, for all cycles is determined whether there is a matter of redundancy. There is a matter of redundancy if two paths between two concepts exist where the products of all minimum and maximum cardinalities of both paths along the cycle are equal. Following our example in figure 5.3, path 'car-axle-wheel' holds $2 * 2 = 4$ as minimum and maximum cardinality which equal the minimum and maximum cardinality of path 'car-wheel'. Thus, one of these relations can be removed without loss of information.

Semantic errors. A semantic error occurs when the model does not correctly represent the facts to represent. Now we need a reference model to decide whether there is matter of a semantic error. Assume both models in figure 5.10, where (a) is the reference model and (b) is the student's model. In model (a) we can determine a path: car - axle - wheel, with corresponding cardinalities (2,2) and (4,4). Using this model, we are able to determine which wheel is connected to which axle for example. In model (b), this determination is not possible, so model (b) does not enclose all information needed in relation to reference model (a). In other words, model (b) is not

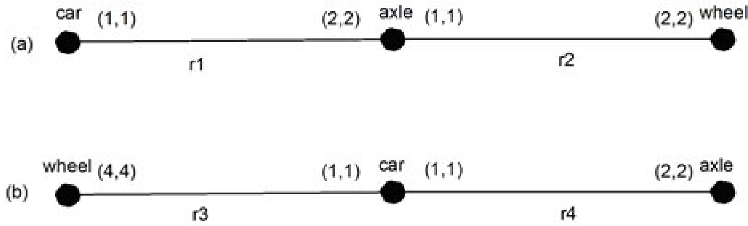


Figure 5.10: A semantic error

semantically isomorph with respect to the reference model (a). Again this type of analysis is performed using a DFS algorithm.

So far, we have ignored the problem of which concept and relation as part of the central model is related to which class, relation or attribute in the student's model. One possibility is asking the student to map each class, relation and attribute as part of the student's model to the concepts and relations of the central model. Another solution will be mentioned in section 5.4.

5.4 Related work

We have described the results of our literature study about related ITSs in chapter 2. As far as we know, there is no literature about ITSs supporting different modeling languages.

The framework described in this chapter assumes a *model-based* approach [88], i.e. the framework assumes a central model or author's model. We think that it is possible, and sometimes desirable, to add extra constraints to express additional semantics. In that case we have a mixed approach, i.e. model- and constraint-based.

Notice that by introducing the framework, we do not have solved the problems mentioned in chapter 2.5, i.e. generations of sample solutions may be too expensive (if there are many different ways of solving the problem), checking the correctness of a correct student's solution which does not match one of the sample solutions completely is difficult, and generating feedback for a partly finished solution matching several sample solutions is difficult [144]. We think that by combining the strategies model-based and constraint-based, these problems can be partly solved. Because many types of modeling tasks are ill-defined, as for example modeling a UML class diagram, we will probably never totally solve these problems. Ill-defined

modeling tasks can be made more well-defined by formalizing the domain of interest, the task description, and the goal to reach. This could be a solution for simple modeling tasks, within a first course about modeling, but is undesirable for more real modeling tasks.

In this chapter, we have restricted ourselves to simple UML class diagrams consisting of classes, associations, multiplicities, and attributes. For example association classes, aggregations, compositions, and generalizations did not have our attention. How to translate these constructs into the central model language is not discussed. These translations are possible [15, 53].

Meta-model errors. Analyzing meta-model errors is an active research field in computer science. We describe some examples.

Egyed [44] reported about the use of abstraction rules for UML-class diagrams. These rules can be used for reverse engineering as well as for checking on consistency. One mechanism called *relation abstraction* is described which can be used to check on consistency of cardinalities. This mechanism looks like the mechanisms discussed in section 5.3.3.

Moha et al. [107, 108] reported about a technique to automatically detect and correct software architecture defects in object-oriented design. The technique distinguishes:

- *Anti patterns* – which are literary forms describing commonly applied solutions to design problems that generate decidedly negative consequences.

- *Design defects* – which are errors in the design of the software that come from the absence or the bad use of design patterns.

- *Code smells* – which are structures in code that suggest the possibility of refactoring.

The aim of the research is to formally describe these architectural defects and to develop mechanisms to detect and correct them automatically. Similar research has been done by Fradet et al. [47]. They have developed a framework for the definition of multiple view architectures and techniques for the automatic verification of their consistency. Other work has been done by Backstrom et al. [11]. They state that guidelines for model structure, naming conventions and other model rules are often provided in practical contexts developing models. Meta-models can be used to describe these guidelines and make it possible to automatically check whether a UML-model satisfies the guidelines or not and produce correction suggestions.

Balaban et al. [14] have described *correctness* problems and the detection of these problems in UML class diagrams. They distinguish between inconsistency, redundancy, and abstraction errors. They describe a pattern-based approach for the detection of these problems within class diagrams, and for providing explanations and repair advices. Another approach to detect inconsistencies in UML class diagrams is described by Miloudi et al. [99]. In this approach a UML class diagram is translated into a formal specification (Z notation) to uncover most of the UML inconsistencies published to date. A similar approach based on first-order logic is described by Satoh et al. [133].

Semantic errors. While there is much literature about analyzing meta-model errors, literature about analysing the semantics of a model is rare. In chapter 2 we have reported about two main approaches, namely model- and constraint-based. There, we have reported about the work of Mitrovic et al. [101, 102, 118].

Striewe and Goedicke [144] describe a *rule-based* approach. In this approach, each rule represents a desirable or undesirable feature of a correct solution and triggers feedback when this feature is not found or found, respectively. This approach assumes that human tutors would check a student's solution by asking themselves questions like 'Does the solution contain an element representing X?' or 'Is there an association between the elements representing X and Y?'. Depending on the answers of these questions, the student's solution is marked as right or wrong. The rules are implemented by queries for checking on the existence of diagram elements based on their type (for example a class or an association), the existence of diagram elements based on the value on their attributes, and the existence of tuples of elements based on the connections between them (for example, a class having an operation that takes another class as parameter type). These queries could be combined into more complex queries by using logical operators. Two sets of rules are distinguished. One set contains the rules specific for a particular task, for example using element names taken from the task description. The other set contains generic rules that are concerned with general features of correct UML, for example missing names and cardinalities. We remark that in fact this rule-based approach is equal to the constraint-based approach [118].

Maoz et al. have described a tool called CDDiff for determining the semantic difference between two UML class diagrams [89]. The semantics of a class diagram are given in terms of object models, consisting of sets of objects and the relationships between these objects. The input of the *cddiff* operator consists of two class diagrams and the operator outputs a set of *diff*

witnesses. Each of these *diff witnesses* is an object model that is possible in the first class diagram, but is not possible in the second class diagram. As the output set may be infinite, a bounded version $cddiff_k$ can be used which only includes object models where the number of object instances is not greater than k. Operator *cddiff* can be used to compare two class diagrams and decide whether one class diagram semantics include the other class diagram semantics (the latter is a refinement of the former), are they semantically equivalent, or are they semantically incomparable (each allows instantiations that the other does not allow). Maoz et al. use the tool to analyze the semantic evolution of a class diagram in a project.

To overcome, for example, the problem of which nouns and verbs from the text might be used, the students can highlight a word or phrase that corresponds to a construct as they add them to the diagram [145]. The highlights are color codes for marking classes, associations and attributes for example. Using this approach, a student is free in naming entities, relations and attributes.

5.5 Conclusions

In this chapter, we have introduced a framework for providing feedback in an ITS for modeling that supports several modeling languages. The framework consists of three main parts: a central model representing the ideal solution of a certain exercise, a set of language transformers, which transforms a model represented in a certain language into an instantiation of the language used in the central model, and a set of analysis functions. The framework distinguishes three types of analysis: syntactic, meta-model and semantic analysis. The advantage of our framework is the reuse of the task description and de central model for several model languages.

Chapter 6

From ill-defined to well-defined tasks

6.1 Introduction

Generating valuable feedback during modeling activities implies a well-defined domain of interest, a well-defined modeling language as well as a well-defined task. An example of such an activity is rewriting a system of linear equations to a solution, for which a complete schema exists, i.e. the modeling language, initial state, goal state, constraints, rules and algorithm are all well known.

Within computer science however, this is often not the case. Some problems are intrinsically ill-defined, for example developing a design model for a complex information system. For this category of problems, there is no complete schema of how to develop such a model, i.e. the initial and goal state are often ill-defined, the constraints are often unclear and there is no algorithm of how to develop such a model (instead, only some rules of thumb exist).

For other modeling activities, schemata are (partially) known, but these are often hardly taught. Many of today's computer science courses introduce and explain their topics without mentioning their underlying formal methods [84]. As a result, it remains unclear how to construct a program, an algorithm, a data structure, a model, etcetera. For example, introductory courses on data structures and algorithms are often limited to the common

This chapter is based on: Passier, H. and Heeren. B. (2011). Modeling XML content models explained. Technical report, Department of Information and Computing Sciences, Utrecht University, 2011. [121]

data structures and accompanying algorithms, and how to use these, but how to develop algorithms on new data structures is not always explained. Instead, teaching methods and engineering approaches are used that mainly rely on inspiration and intuition, and this does not always work out well. As a result, students are often not sufficiently aware of what to do and why: they need and ask for more guidance in terms of 'how to do' a particular task.

An example of this category is modeling an XML content model. Although the similarities between schema languages and regular expressions are well-understood, books and teaching material do not use this to their advantage. Typically, a number of (small) examples is given, but without an explanation of how the resulting content model was found. We are not aware of books on XML that introduce regular expressions to provide a deeper insight into content models, or a systematic way to model XML content.

For modeling XML content models, we have developed a complete schema consisting of a domain, an initial state, a goal state, constraints, rules to manipulate expressions in the domain, and a strategy. The strategy consists of two parts. For one category of XML content models, precise models (describing exactly the set of allowed sequences of XML elements, but nothing more), we have developed an algorithm. For a second category of XML content models, correct models (describing at least the sequences of XML elements we want to have), we have developed some rules of thumb.

In this chapter, we present a systematic approach based on rewriting regular expressions [10] that helps students in constructing content models as part of an XML schema. By establishing a link between XML schema languages and regular expressions, it becomes much easier to reason about content models, and to manipulate these models. Rewrite rules on regular expressions pave the way for a stepwise derivation of a content model.

Developing a schema for solving a problem forces a teacher to be explicit about all terms and the semantics of a particular domain, for example the goal that has to be reached, how progression of the solving process can be measured, and which rules may be used in which order. Such a schema is a necessary condition for feedback generation.

Another advantage of such a schema is that students learn how to approach modeling tasks systematically. During lectures about XML content modeling, we have made the following observations about students asked to give a suitable XML content model for some XML instance document:

- They have difficulties to get started: they need assistance with the first steps in constructing a complex content model, or the evaluation of such a model;

- The process of finding a model is not structured, and involves a lot of trial-and-error;
- Resulting models are too liberal (the schema accepts too many XML documents), or even incorrect (parts are missing in the schema);
- Resulting models are not deterministic;
- Students are unable to evaluate a generated model on correctness and precision.

Similar observations can be made about learning material for other XML-related languages, such as the navigation and selection language XPath, and the transformation languages XQuery and XSLT. The use of a schema of how to develop an XML content model helps students in learning to model complex XML content models. We have tested our modeling approach on a group of students. The test clearly shows that the approach can help students in learning to model complex XML content models. Test and interview results are given in the last part of this chapter.

The chapter is structured as follows. Section 6.2 gives an introduction to DTDs and regular expressions, and presents rules to rewrite these expressions. Section 6.3 then explains how to make a content model deterministic, a requirement of the DTD language. Sections 6.4 and 6.5 define strategies for precise and correct content models, respectively. We then discuss our small-scale experiment (section 6.6). The last sections give related work, draw conclusions, and give directions for future work.

6.2 DTDs and Regular Expressions

We start with a comparison of DTDs and regular expressions (REs), followed by a formal definition of the language that is generated by a RE. In the final part, we give some rewrite rules for manipulating expressions.

6.2.1 Syntax

A DTD lists all the elements that can be part of an XML document by means of element declarations, such as:

```
<!ELEMENT book (title, author+, chapter+)>
```

The content model of an XML element specifies which child elements may occur, and in which order. In our example, `(title, author+, chapter+)` is the content model of element `book`. Each element `book` must have a

Construct	DTD notation	RE notation
empty set		$\emptyset$
empty string	*EMPTY*	ϵ
alphabet	element names	atoms
sequence	R, S	RS
choice	$R \mid S$	$R \mid S$
zero or one	$R?$	$R?$
zero or more	$R{*}$	R^*
one or more	$R{+}$	R^+

Figure 6.1: Syntax of DTDs versus REs

`title` element, followed by one or more `author` elements, and one or more `chapter` elements. In the remainder of this chapter, we choose DTD as our schema language, but our approach works for other languages as well (such as XML-Schema).

The syntax of a content model differs slightly from standard RE notation: Figure 6.1 shows the correspondence between the notations. There is no counterpart of the empty set construct in DTD notation. Furthermore, ϵ concisely denotes the empty string. Also observe how the commas are dropped for sequences (since the atoms of an RE are generally assumed to be single characters, unless otherwise noted). For reasons of presentation, we adopt the RE syntax in this paper, without the $\emptyset$ construct.

We use R, S, and T to represent arbitrary REs, and $a, b, c, \ldots$ for the atoms in our examples. The standard precedence levels apply: the unary operators (?, *, and $+$) bind stronger than sequence, which binds stronger than choice. Parentheses are used to group expressions. Hence, the expression $ab^* \mid c$ is interpreted as $(a(b^*)) \mid c$, and not $a(b^* \mid c)$ or $(ab)^* \mid c$.

6.2.2 Language

An RE describes a possibly infinite set of sentences, which we call the language generated by that expression, denoted by $\mathcal{L}(\cdot)$. This can be defined

$$
\begin{array}{rcll}
R \mid (S \mid T) & = & (R \mid S) \mid T & \text{(1a)} \\
R \mid S & = & S \mid R & \text{(1b)} \\
R \mid R & = & R & \text{(1c)} \\
R(ST) & = & (RS)T & \text{(1d)} \\
\epsilon R & = & R & \text{(1e)} \\
R\epsilon & = & R & \text{(1f)} \\
\\
R? & = & \epsilon \mid R & \text{(2a)} \\
R^* & = & \epsilon \mid RR^* & \text{(2b)} \\
R^+ & = & RR^* & \text{(2c)} \\
\\
RS \mid RT & = & R(S \mid T) & \text{(3a)} \\
RT \mid ST & = & (R \mid S)T & \text{(3b)} \\
R^*R & = & RR^* & \text{(3c)} \\
(RS)^*R & = & R(SR)^* & \text{(3d)}
\end{array}
$$

Figure 6.2: Rewrite rules on REs

inductively as follows [73]:

$$
\begin{array}{rcl}
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(a) & = & \{a\} \\
\mathcal{L}(ST) & = & \mathcal{L}(S)\mathcal{L}(T) \\
\mathcal{L}(S \mid T) & = & \mathcal{L}(S) \cup \mathcal{L}(T) \\
\mathcal{L}(S?) & = & \mathcal{L}(\epsilon \mid S) \\
\mathcal{L}(S^*) & = & (\mathcal{L}(S))^* \\
\mathcal{L}(S^+) & = & \mathcal{L}(SS^*)
\end{array}
$$

Here, concatenation of two sets, written XY, is short-hand notation for $\{ xy \mid x \in X, y \in Y \}$. The star-closure of a set, X^*, equals $X^0 \cup X^1 \cup \ldots$, where:

$$
\begin{array}{rcl}
X^0 & = & \{\epsilon\} \\
X^{n+1} & = & XX^n
\end{array}
$$

Similarly, R^n is used as shorthand notation for a sequence of n occurrences of regular expression R.

These definitions provide the proper foundation to reason about and manipulate REs. In forthcoming sections, we use this to evaluate content models on correctness.

6.2.3 Rewrite rules

Figure 6.2 presents a list of rewrite rules that operate on REs. The first set of rules (1a - 1f) expresses some basic properties of the choice and sequence

$$R^n \Rightarrow R^* \quad (\textit{if } n \geq 0) \qquad (4a)$$
$$R^n \Rightarrow R^+ \quad (\textit{if } n \geq 1) \qquad (4b)$$
$$R^*S^* \Rightarrow (R \mid S)^* \qquad (4c)$$

Figure 6.3: Directed rewrite rules

operators: choice is associative, commutative, and idempotent, whereas sequence is associative and has ϵ as its unit. Soundness of these rules follows straightforwardly from the language generated for both sides of the equation. The rules for associativity (1a and 1d) are typically performed implicitly, and parentheses are dropped accordingly.

The second set of rules (2a–2c) defines a translation for each of the cardinality operators. These rules show that all occurrences of $R?$ and R^+ can be removed from an expression. On the other hand, R^* can be expanded one step.

The last set of rules is for making expressions deterministic, which is discussed in the next section. We have rules for left factoring (3a), and right factoring (3b). Rule 3d (and 3c as a special case) helps in rearranging expressions involving R^* (under the right circumstances these can be shifted to the right). More rewrite rules can be added to the collection, for example by combining existing ones. When modeling XML content with REs, it is convenient to have a rich set of rules that covers common patterns. Note that the rules in figure 6.2 can be applied in both directions (i.e., also from right to left) because both sides are equal.

When modeling XML content, one typically uses the cardinality operators to reduce the size of the model. For example, $a \mid aa \mid aaa$ can be written as a^+, which is far more concise. The price we pay for this reduction in size is a loss of precision: the latter expression now also accepts $aaaa$. Figure 6.3 shows two more rewrite rules for the introduction of cardinality operators. These rules are directed from left to right.

Semantically, these directed rules extend the language that is generated. To specify this property, we introduce a partial ordering between REs: $R \leq S$ if and only if $\mathcal{L}(R) \subseteq \mathcal{L}(S)$. A directed rewrite rule $R \Rightarrow S$ must satisfy $R \leq S$, and indeed, the rules of figure 6.3 have this property.

6.3 Removing non-determinism

XML is defined to be compatible with SGML, and as a consequence, content models of DTDs have to be deterministic. A content model is deterministic if an XML processor can check a document against a DTD without looking for-

ward in the document (i.e., inspecting only the current element). Generally, there are exactly two situations in which non-determinism occurs [156]:

1. A content model contains $R \mid S$ and the sets of element names that can start a sequence in $\mathcal{L}(R)$ and $\mathcal{L}(S)$ are not disjoint. For example, $ab \mid ac$ is not deterministic because the set of starters (known as the *first* set [25]) is $\{a\}$ for both alternatives.

2. A content model contains $R?$, R^*, or R^+, and the set of element names that can start a sequence in $\mathcal{L}(R)$ is not disjoint with the set of names that can follow in this particular context (the *follow* set [25]). An example of such a non-deterministic expression is $(ab)^*ac$.

6.3.1 Strategy for removing non-determinism

We now present a strategy for the stepwise removal of non-determinism: rewrite problematic sub-expressions (one of the two situations described above) until we have reached a deterministic expression. We discuss the two situations.

Situation 1.

Given is a sub-expression $R \mid S$ with at least one element that is starter of R and S. Let this element be a. The non-determinism can be removed in three steps.

(a) Rewrite R and S until element a is the first of a sequence. This involves expanding cardinality operators (2a–2c), removing ϵ in sequences (1e), and distributing sequence over choice (3b). Rules 1c, 3c, and 3d (and variants for the other operators) can provide a shortcut.

(b) If needed, rearrange alternatives (1b) so that the sequences starting with a are adjacent.

(c) Apply the factorization rule 3a. In some cases, an ϵ has to be introduced first (1f).

Example Applying the above strategy to the non-deterministic expression $(ab)^+ \mid bc \mid abc$ results in:

$$\begin{array}{rcll} (ab)^+ \mid bc \mid abc & = & ab(ab)^* \mid bc \mid abc & (2c) \\ & = & ab(ab)^* \mid abc \mid bc & (1b) \\ & = & ab((ab)^* \mid c) \mid bc & (3a) \end{array}$$

Example The top-level alternatives in the following example make the expression non-deterministic:

$$\begin{aligned}(a \mid b)a \mid a &= aa \mid ba \mid a && \text{(3b)}\\ &= aa \mid a \mid ba && \text{(1b)}\\ &= aa \mid a\epsilon \mid ba && \text{(1f)}\\ &= a(a \mid \epsilon) \mid ba && \text{(3a)}\end{aligned}$$

The last two steps are a good candidate for adding a new, derived rewrite rule to the collection: $RS \mid R = R(S \mid \epsilon)$.

Situation 2.

The second case is more complicated because some REs cannot be transformed into a deterministic form [17, 24, 25]. Examples of such expressions are $(ab)^*(ac)^*$ and $(ab)^*a?$. Note that deterministic REs should not be confused with deterministic finite-state automata (DFA) [66], another well-known formalism. Every RE can be transformed into an equivalent DFA, and the other way around. An RE constructed from a DFA, however, is not automatically deterministic.

Suppose we have some sub-expression $R?$, R^*, or R^+, which has element a as a starter. Furthermore, a is also in the follow set of the sub-expression at hand. We proceed by case analysis on the operator used.

(a) For content model $R?$, we remove the operator by applying rule 2a, resulting in the alternative $\epsilon \mid R$. Then, this alternative should be combined with its context, for instance by using distribution (rule 3b, from right to left). Eventually, we arrive at a situation 1 problem. This case also covers non-deterministic expressions of the form $\epsilon \mid R$ (instead of $R?$) that have a non-disjoint follow set.

(b) At its best, removing non-determinism involving R^* can be done with rules 3c and 3d. In some cases, expression R or its context needs some rewriting before these rules are applicable. If this does not work (e.g. because it is impossible), then two possibilities remain to deal with the situation:

- Make the expression less precise, and extend the language generated by the RE.
- Introduce an extra level in the XML tree, and circumvent the ambiguity altogether.

(c) In case of R^+, we use rule 2c. This reduces the problem to the case for R^*.

Example The following derivation illustrates the case for an optional part:

$$\begin{aligned} (ab)?a &= (\epsilon \mid ab)a && \text{(2a)} \\ &= a \mid aba && \text{(3b and 1e)} \\ &= a(\epsilon \mid ba) && \text{(3a and 1f)} \end{aligned}$$

Example Consider the regular expression $(ab)^*(ac)^*$, for which there is no equivalent deterministic RE. We can opt for a less precise (but deterministic) model:

$$\begin{aligned} (ab)^*(ac)^* &\leq (ab \mid ac)^* && \text{(4c)} \\ &= (a(b \mid c))^* && \text{(3a)} \end{aligned}$$

Alternatively, we can decide to introduce an extra level by defining new elements. Suppose that the model $(ab)^*(ac)^*$ belongs to an element `abacs`. We can split the model into two parts, resulting in the following element declarations:

```
<!ELEMENT abacs (abs, acs)>
<!ELEMENT abs (a,b)*>
<!ELEMENT acs (a,c)*>
```

The new element names are `abs` and `acs`. Notice that introducing extra levels is not related to rewriting REs.

6.3.2 Normal form for content models

Besides the requirement that content models have to be deterministic, an ϵ should not be part of a composite content model according to the DTD language specification. This means that in the end, ϵ's have to be removed, which is fairly simple (e.g., with rules 2a, 1e, and 1f). This gives us a normal form for content models.

Definition A content model M is in XML normal form (XNF) if and only if M is deterministic, and no ϵ is present in M (except when M itself is ϵ).

Notice that an XML content model in XNF is not unique. For example, we can always rewrite $a(b \mid c)$ into $a(c \mid b)$ applying rule 1b. Reaching XNF is not a goal of its own, but rather a final step after the strategies presented next (section 6.4 and section 6.5).

6.4 Precise content models

We now turn our attention to deriving a content model from some instance document. A content model should not be made too liberal without careful thought: after all, schema languages are used in the first place to reject documents and to spot inconsistencies. We start by considering precise content models only: a precise content model contains exactly those sequences of child elements that we want to have, and no others.

Definition Let M be a content model, and X a set of sequences of child elements. Then M is a precise model for X if and only if $\mathcal{L}(M) = X$.

Obtaining a precise model for some XML content is rather easy. First, we write down all sequences of child elements for some particular element that appear in the instance document. These sequences are the choices of the starting model. For example, suppose we have:

```
<recs>
  <rec>
    <a/>
    <b/>
  </rec>
  <rec/>
  <rec>
    <a/>
    <b/>
    <a/>
    <b/>
  </rec>
  <rec>
    <a/>
    <b/>
  </rec>
</recs>
```

This instance document contains four sequences of child elements for element `rec`. Hence, the starting model is $ab \mid \epsilon \mid abab \mid ab$. We call such a first approximation of a precise content model the starting form (SF).

The next step is to rewrite the content model in starting form, and turn the model into XNF without loosing precision. A strategy for this step is discussed next.

6.4.1 Strategy for precise content models

The strategy described in section 6.3 can turn any content model into XNF. However, if we start with a model in SF and look for a precise model, a much simpler strategy is sufficient. More specifically, cardinality operators are not present in the starting model, nor are they introduced during rewriting (since that would make the model no longer precise). The strategy for precise content models is as follows:

Input: An XML content model in SF.
Output: A precise XML content model in XNF.

Step 1. Remove redundant choices of the form $R \mid R$ by applying rule 1c. If the duplicate choices are not adjacent, then change the order (rule 1b).
Step 2. Remove situation 1 type of non-determinism (for sub-expressions of the form $R|S$) by repeating the strategy of section 6.3.1 until the model is deterministic.
Step 3. To reach XNF, we remove all occurrences of ϵ. For this, we apply rule 2a, from right to left.

The strategy for precise content models returns canonical models (up to the order of choices). The order in which the rules are applied does not influence the result. Removing duplicates early (step 1) helps to shorten the derivations. Also note that the size of the final model is never larger than the original model.

Example We continue with the starting form for element `rec`, introduced earlier in this section. We rewrite its model into a precise model in XNF.

$$
\begin{aligned}
ab \mid \epsilon \mid abab \mid ab &= ab \mid ab \mid \epsilon \mid abab && \text{(1b)}\\
&= ab \mid \epsilon \mid abab && \text{(1c)}\\
&= \epsilon \mid ab \mid abab && \text{(1b)}\\
&= \epsilon \mid ab\epsilon \mid abab && \text{(1f)}\\
&= \epsilon \mid ab(\epsilon \mid ab) && \text{(3a)}\\
&= \epsilon \mid ab(ab)? && \text{(2a)}\\
&= (ab(ab)?)? && \text{(2a)}
\end{aligned}
$$

6.4.2 Using the strategy

The strategy for precise models should be used if the number of choices in the SF is relatively small, and rewriting the model in a precise way is still manageable. Also use the strategy if a precise model is needed, i.e., the context demands a content model representing exactly the sequences of child elements in the SF and no other ones.

Precise models are not always desirable in practice, however, because models quickly become too verbose. For example, if we assume that a book record has exactly one ISBN but multiple authors and chapters, a content model with cardinality operators $(isbn, author^+, chapter^+)$ is more appropriate than a model without these operators.

The precision of a content model cannot be tested with a validating parser. The only approach here is to manually generate the language of the content model (see section 6.2.2) and check whether the sequences of child elements we want to have are the same as the language of the content model.

6.5 Correct content models

A correct content model contains at least the sequences of child elements we want to have, and possibly more.

Definition Let M be a content model, and X a set of sequences of child elements. Then M is a correct model for X if and only if $\mathcal{L}(M) \supseteq X$.

For instance, $(ab)^*$ is a correct content model for $\{\epsilon, ab, abab\}$, but not a precise model since $ababab \in \mathcal{L}((ab)^*)$. Correct models are generally more concise than precise models: the trade-off is that they can be more liberal than needed.

Smaller models show the structure more clearly. For example, $(a \mid b)^*$ is equal to $(a^*b^*)^*$, but the first one is (arguably) simpler. Minimizing the size of an expression should not be the only goal though. A model that allows everything (e.g. $(a_1 \mid a_2 \mid \ldots \mid a_n)^*$ where $a_1 \ldots a_n$ are all existing elements, which can be abbreviated to ANY) is concise, but defeats the purpose of writing content models. The challenge is to find the right balance between conciseness and precision. For this, expert knowledge about the domain being modeled is needed. For example, $chapter^+$ is reasonable for a book record, whereas $isbn^*$ is questionable. Such decisions cannot be made automatically by a strategy.

6.5.1 Strategy for correct content models

We now present a strategy for correct (but not necessarily precise) content models. This strategy introduces cardinality operators during rewriting. As a rule of thumb, cardinality operators should be introduced early on, and before factorization, because the initial model in SF best exposes the

replicated parts. The introduction of cardinality operators can lead to non-deterministic models, for which we use the strategy described in section 6.3 to remove this non-determinism.

Input: An XML content model in SF.
Output: A correct XML content model in XNF.

Step 1. Remove redundant choices of the form $R \mid R$ (rule 1c). Change the order of alternatives if needed (rule 1b).

Step 2. Search for opportunities to introduce cardinality operators, and make sure that this is appropriate in the underlying domain. Find all choices that can be combined, and place these next to each other (rule 1b). If the ϵ alternative is not present, rewrite all choices to R^+ (rule 4b); otherwise, use rule 4a. Afterwards, duplicate alternatives can be removed (rule 1c). Sometimes, parts have to be rewritten before the cardinality operators can be introduced.

Step 3. If no more cardinality operators have to be introduced, bring the expression into XNF by applying factorization and removing ϵ's. The details of this procedure are discussed in section 6.3.

Example Consider the model $ab \mid abab \mid abc \mid \epsilon$, which is in SF. We identify three out of four alternatives as instances of $(ab)^*$, i.e., zero or more occurrences of ab. Rewriting the term then proceeds as follows:

$$\begin{array}{rcll}
ab \mid abab \mid abc \mid \epsilon & = & \epsilon \mid ab \mid abab \mid abc & \text{(1b)} \\
 & \leq & (ab)^* \mid (ab)^* \mid (ab)^* \mid abc & \text{(4a)} \\
 & = & (ab)^* \mid abc & \text{(1c)} \\
 & = & \epsilon \mid ab(ab)^* \mid abc & \text{(2b)} \\
 & = & \epsilon \mid ab((ab)^* \mid c) & \text{(3a)} \\
 & = & (ab((ab)^* \mid c))? & \text{(2a)}
\end{array}$$

The resulting model is in XNF. The step in which we give up precision and introduce $(ab)^*$ is made explicit in the derivation, and this is where domain knowledge is required. We could have decided to also rewrite the sub-expression abc into $(ab)^*c$, which would lead to the more concise (but less precise) content model $(ab)^*c?$.

6.5.2 Using the strategy

The strategy for correct models should be used if the number of choices in the SF is large, and a precise content model would be too verbose. The correctness of a final content model can be tested using a validating parser: this

parser checks all sequences of child elements in the instance XML-document against the content model, and it will complain if the model is incorrect.

Our strategies are also useful if we start with an informal description of a content model, instead of a starting form. For example, in a chess game, white and black alternate moves, and white has the opening move. These requirements could be translated into the model $(white, black)^+, white?$. The strategy for removing non-determinism (situation 2, case b) suggests to make the model less precise, or to introduce an extra level.

6.6 Experiment and validation

Five students have participated in a small experiment consisting of a pretest, an online lecture of two hours, a posttest, and an interview. These students have followed the regular bachelor course on XML. The course introduces schema languages, but does not explain any method for modeling content models. All students, except for one, have some basic knowledge about REs or propositional logic.

During the lecture, the relation between content models and REs, the syntax of REs, the language generated by an RE, the rules for rewriting REs, non-determinism and how to remove this, and strategies for modeling precise and correct models were discussed. Students were asked to practice with some exercises. After the lecture, the students had access to the lecture sheets (including examples and exercises).

The pre and posttest consisted of nine questions about modeling precise and correct content models, and removing non-determinism by rewriting or by introducing extra levels in the XML-tree. In the posttest, four questions were repeated from the pretest. The pre and posttest were marked after the students were interviewed: answers were either correct (1 point) or wrong (no score).

6.6.1 Results

All students scored one or two points higher on the posttest with respect to the pretest; the mean score increased from 4.6 to 6.0, where the maximum score was nine points. Furthermore, we observed a shift in the kind of mistakes. In the pretest, students often produced models that are too liberal for the XML instance document, or models that are not deterministic. Typically, only a final model was given. In the posttest, intermediate steps were given by the students, although not always successfully. Typical mistakes were the incorrect application of the distribution rule and the empty content

missing in the starting form. In addition, we observed an over-carefulness in introducing cardinality operators.

In the interview, the students were asked to what extent the strategies helped in finding precise and correct models. The students reported that the method was particularly useful to get started with complex models. In the pretest, most used a trial-and-error approach.

One student is using the method in daily practice. The students also stated that the approach provided a better understanding of precise versus correct models. As a consequence, they think more carefully about introducing cardinality operators.

The participants did not find the method difficult to learn, but they indicated that more practice is needed for applying the rewrite rules and strategies without errors. The students also agreed with the claim that formal methods are lacking in computer science education. They replied: *"formal methods help me in solving problems"*, *"I often miss a systematic approach"*, and *"it helps me in how to begin"*.

6.6.2 Discussion

We are careful not to draw strong conclusions based on the tests and the interview. For this, an experiment on a larger scale is needed. We also acknowledge that the students were encouraged to practice with modeling XML content between the pre and posttest, and to study the new material, which also contributes to the improved scores for the posttest. Nevertheless, the students generally welcome the use of formal methods for a practical purpose, and our approach has some clear advantages from an educational point of view.

The first advantage is that it stimulates students to write down a stepwise derivation, and not just a final answer. Once students become more familiar with rewriting models, some trivial steps can be safely skipped. A stepwise approach helps in decomposing a complex task, which is particularly helpful to get started. We observed that many errors during rewriting were not noticed, partly because the students are not accustomed to check their answer. Such a sanity check deserves more attention in teaching the method. We expect that an ITS by which students can practice modeling XML content models would be valuable here.

A second advantage is that students are much more aware of the strictness of a content model, and the consequences of introducing cardinality operators. This aspect of schema languages is often overlooked in teaching material on XML.

6.7 Related Work

Systematic approaches to problem solving play an important role in education. These approaches are often based on three components: knowledge about the domain, means to reason with that knowledge, and a strategy or procedure to guide that reasoning [27, 96]. Our approach is based on making the rewrite rules, and the stratgey for using these rules, explicit.

In computer science education, the incorporation of formal methods is strongly suggested by scientific societies such as ACM/IEEE, and many influential scientists [97]. Students employing formal methods during analysis and specification produce more correct, concise, and less complex models [138]. In many curricula, however, formal methods are treated solely as a separate subject to study [84]. Wing et al. [159] advise to weave the use of formal methods into existing courses, making it an additional problem solving technique. We think that our approach is a good example of this advice.

There is an extensive literature about the algorithmic inference of XML content models, and about dealing with non-determinism [18]. These algorithms often involve the construction of finite-state automata, which makes them more difficult to carry out by hand. We are not aware of other approaches that aim at manually deriving models, at the level of an undergraduate course.

6.8 Conclusions

We have described a complete schema for modeling XML content models. We have shown that the rewrite rules and strategies for regular expressions, as part of the schema, help students in understanding XML content models, and guide in the stepwise construction of such a model. The approach makes a sharp distinction between precise and correct models. The first results from using the approach in practice are promising: students appreciate the use of formal methods for solving practical problems. More importantly, they produce better XML content models.

Chapter 7

Epilogue and future work

7.1 Summary

In this thesis, we investigated a number of aspects of feedback generation during modeling activities in Intelligent Tutoring Systems (ITSs).

In chapter 1, we described the research context and what we mean by modeling and feedback. We presented some examples of modeling exercises and described the lack of valuable feedback during practicing modeling. In chapter 2, we discussed the main concepts and functions a feedback mechanism consists of and we explored these concepts and functions in the context of feedback in modeling education. After that, we presented the framework. The framework is used to contextualize the research questions. We discussed related work on ITSs for modeling education and presented four research questions.

In chapter 3, we answered the question of how we can produce feedback on a well-defined task, namely solving a system of n linear equations with n variables. Feedback is produced about syntactic mistakes, semantic mistakes, and the (lack of) progress towards a solution. This question is answered by describing a framework for solving a system of linear equations using the substitution method. Solving a system of linear equations is a type of model transformation.

It is shown how structural information in data and term-rewriting can be used to provide semantically rich feedback to a student. The framework assumes a well-defined domain with semantics, a set of rewrite rules, a well-defined goal to reach, and a set of progress indicators to determine the distance between the goal and the current situation. The domain structure consists of a system of linear equations, where each equation is built up of expressions. Following the structure of the domain, rewrite rules are distin-

guished on the level of a system of equations, on the level of an equation, and on the level of an expression. We described how we can detect a rewrite on the system of equations level, on the equation level, and on the expression level. The hierarchical approach to determining which rewrite rule has been applied, allows us to pinpoint precisely, in many cases, which mistake has been made and to produce valuable feedback for a student. A set of progress indicators informs a student about the progress towards a solution. These indicators are independent of the rewrite rules. A next step or hint can be produced based on the substitution method as solving method.

The main ideas behind the analysis for feedback are reusable in other domains. The implementation of the Equation Solver is not reusable. For a new domain with its own rewrite rules, we have to build a completely new tool. This problem is solved in later projects part of the IDEAS project.

In chapter 4, we answered the question of how we can analyze several properties of a domain ontology and course structure during the authoring process and thereby allow for more flexibility in authoring processes. We showed how schema analysis techniques can be used to detect if certain properties hold for a course. These properties are completeness (Are all concepts that are used in the course defined somewhere?), timeliness (Are all concept used in the course defined on time?), recursive concepts (Are there concepts defined in terms of itself?), correctness (Does the definition of a concept used in the course correspond to the definition of the concept in the domain ontology?), synonyms (Are there concepts with different names but exactly the same definition?), and homonyms (Are there concepts with the same name, but different definitions?).

In chapter 5, we made a start at answering the question of how we can support different data modeling languages in an ITS without re-developing important parts of the ITS for each new data modeling language introduced. To answer this question, the outline of a framework is presented that (1) generates feedback about syntactic errors of the language used and the model's intended semantics, and (2) is able to generate feedback for several model languages. The framework distinguishes between three types of errors, namely syntactic errors, meta-model errors, and semantic errors. The framework consists of a central model, or standard solution, mappings for translating a student's solution into the language of the central model and vice versa, and analysis functions for analyzing the student's solution on meta-model errors and semantic errors. The translations between the student's solution and the central model must satisfy the property of invertability. To be able to satisfy this property, we make use of Chisholm's ontology. The framework is implemented for simple UML class diagrams

consisting of classes, attributes and associations.

In chapter 6, we have answered the question of what is needed to specify a precise task description, i.e. to transform an ill-defined task into a well-defined task. We investigated this by developing a complete schema for modeling XML content models. XML content models are described using the regular expression notation. The schema consists of a syntax description, a set of rewrite rules for manipulating XML content models, a description of the semantics of an XML content model, a strategy of how to construct such models, and descriptions of the start state and the goal state to reach. Furthermore, the strategy distinguishes between two types of models. For models of the first type, precise models, the strategy is an algorithm. For models of the second type, correct models, the strategy consists of rules of thumb. The schema is validated by a small experiment. The results are promising: students appreciate the use of formal methods for solving practical problems and produce better content models.

7.2 Putting things into perspective

Can we produce valuable feedback on all categories of models? The answer is definitely *no*. To be able to produce valuable feedback on a model and the modeling process followed, we need to have a well-defined modeling language, a well-defined task, and a well-defined domain of interest (chapter 2). In most modeling situations, this is not the case. What are the consequences for modeling education and the design of learning technologies?

Recently, Le et al. [86] have described a classification of the degree of ill-definedness of educational problems based on the existence of strategies, the implementation variability for each strategy, and the verifiability of solutions. This classification divides educational problems into five classes. We briefly describe these classes:

1. *One single strategy, one implementation, and solution correctness can be verified automatically* – Problems of this class can be solved with only one single strategy which has only one implementation, and have only one solution. An example of this class is: 'Write a Java statement to sum the numbers 4 and 5. Please fill in the missing operator: `x = 4 _ 5`.'

2. *One strategy with different implementation variants and solution correctness can be verified automatically* – Problems of this class can be solved with one strategy, but this strategy can be implemented in a

number of different ways. Problems in this class can be specified precisely so that the space of solutions is narrowed down to a single strategy, or the input is restricted by solution templates. An example of this class is: 'Write a function to compute the return R on investment X after N years for a fixed interest of Y. Use a FOR-DO loop.'

Solving a system of linear equations following one particular strategy, as for example the substitution method, and modeling a precise XML content model are two other examples of this class, i.e. several sequences of rewrite steps (multiple implementation variants) for these strategies exist.

3. *A known number of typical strategies and solution correctness can be verified automatically* – For this class of problems, the student is free to choose among several known alternative strategies. Each of these strategies can be implemented in a number of different ways. An example of this class is: 'Write a function to compute the return R on an investment X after N years for a fixed interest of Y.'

   Another example is solving a system of linear equations where the substitution as well as the combination method can be used, even in combination.

4. *A great variety of strategies beyond the anticipation of a teacher, where solution correctness can be verified automatically* – These problems are so complex that it may not be possible to enumerate a priori all possible strategies that may be used. In this class, it is often the case that the number of strategies for any of the sub-problems is not known. An example is: 'Develop a calculator to calculate the return on investment.' The space of combinations of design decisions is large. However, the correctness of each solution can be verified using test cases.

   Another example is modeling a UML class diagram, where the domain of interest is presented in a formalized way. There are many ways of how to make such a model, i.e. classes, attributes and associations can be added in several orders. By formalizing the domain of interest, the solution's correctness can be determined.

5. *Multiple strategies and solution correctness cannot be verified automatically* – In this class, solutions to problems cannot be verified automatically. This can occur, for instance, if a criterion for good solutions should be considered such as 'useful' or 'acceptable' by a large number

> of stakeholders. An example of such a problem is: 'Develop the most user-friendly calculator to calculate return on investment.' While the calculation can be verified using test cases, aspects as 'user-friendly' are ill-defined.
>
> Another example is modeling a UML class diagram, where the domain of interest is ill-defined.

Then, Le et al. discuss for each class a number of approaches to build intelligent educational systems producing (valuable) feedback. Feedback on the first category is simple: to check correctness of a solution, the system simply needs to compare the student's solution against a pre-specified value. For the second class, model-based and constraint-based modeling (CBM) techniques are successfully used, often supplemented with a set of buggy rules. Producing feedback for the third class is much harder, but recently some approaches have been developed. Examples are the ITSs developed in the IDEAS project, in which multiple strategies can be detected. Other techniques mentioned are machine learning and (other) soft computing techniques. Furthermore, the model- and CBM-based approaches from class two can be used too, however developing these for problems from this class is very labor intensive and error prone. Solutions for problems in class four are rarely found. The authors report about three approaches based on data mining techniques and knowledge discovery techniques. A disadvantage of these techniques is that they are error prone. For class five problems, which are very hard, no ITSs are known. Instead, e-learning systems supporting for example peer reviews by humans and collaborative argumentation are reported. Disadvantage of these approaches are the asynchronous character, which causes feedback to be delayed, and the costly peer reviews. In conclusion, solving problems in the first three classes can often be supplied with valuable feedback. For problems in classes four and five, this is often very difficult or even impossible.

The classification of Le et al. corresponds with our classification in chapter 2 (see figure 2.2). *Solution verifiability* corresponds with the degree of definedness of the domain of interest. For a well-defined domain of interest, as for example an XML-document, we are able to verify the correctness of the model (the XML content model) against that domain. For an ill-defined domain of interest, as for example an ambiguous textual description of a certain business process, we are not able to verify the correctness of the model (for example a UML class diagram) against that domain. Due to the ambiguous domain description, many interpretations are possible.

The *number of alternative strategies* and the *implementation variability*

of each strategy correspond with the degree of definedness of a task. For tasks which are algorithmic, even when a number of algorithms can be applied in combination, valuable feedback can be generated about the steps a student takes. Examples are solving a system of linear equations and modeling a precise XML content model. For tasks for which only some rules of thumb exist, we are not able to generate valuable feedback about the solving process. Depending on the defindedness of the domain of interest, only the final model can be verified on correctness.

In our classification, a third dimension *definedness of the modeling language* is added. This is important, because only on models expressed in a language with a clear semantics we can produce valuable feedback. On models expressed in, for example, natural language, this is very difficult or even impossible due to the ambiguity of the language.

In our opinion, there are two main directions to solve the problem of undefinedness in ITSs for practicing modeling and as a consequence the ability to generate automatically valuable feedback:

1. *Narrowing the solution space* – For educational purposes, by doing some preparatory work, we can transform an ill-defined domain of interest into a more well-defined domain. This preparatory work is a movement along the axis Domain of interest from ill-defined to well-defined. In practice, this means presenting a description of the domain of interest in a (more) *formalized*, and thus less ambiguous, way. An example is the generation of a textual description of a domain of interest based on an ontology. A sentence as 'The hunter shoots the rabbit with his gun' becomes for example something like 'A rabbit has a gun. A hunter shoots the rabbit. The hunter shoots with the rabbit's gun'. Another example is the restriction on options for naming classes and attributes in a UML class diagram. If only highlighted words in a textual description can be used, the variability of the solution space is considerably narrowed down. By doing this preparatory work, the possibilities for verifying a student's model solution increases at the cost of the authenticity of the task and the complexity of the task. As a consequence, modeling tasks with a formalized domain of interest are especially usable in first year courses.

2. *Narrowing the task space* – For some modeling tasks, there exist a schema of how to develop such a model. Unfortunately, most modeling tasks are ill-defined. For these problems, there is no known algorithm of how to develop such a model in a controlled and stepwise way. Sometimes, only some rules of thumb exist.

In some cases, the task space of a modeling problem can be narrowed by developing a schema describing how such a model can be developed, i.e. a description of the initial state as precondition, the goal state, and a strategy in the form of an algorithm, procedure, or rules of thumb. We have seen an example of this in chapter 6. Until recently, modeling an XML content model was an ill-defined task. Now we have a well-defined schema of how to develop such a model stepwise and in a controlled manner. As a result, modeling a precise XML content model is moved along the modeling task axis from ill-defined to well-defined. This has two considerable advantages. First, modeling a precise XML content model is moved from class four, where we consider a 'trial-and-error' approach as 'a great variety of strategies beyond the anticipation of a teacher', to class two in Le's et al. classification. As a result, we are able to generate automatic feedback on this modeling task. Second, we can now teach students how to develop such a model in an efficient and effective way.

7.3 Future work

We conclude with some suggestions for further work. We see two main directions, namely *narrowing the task space* and the further *development of technology*. Looking to figure 2.3, the first one focuses especially on the task ontology, the second one on the feedback engine, the player, and the author tool.

Narrowing the task space. The discussion so far makes clear that 'no schema' means 'no feedback' and 'the more defined a schema', 'the more valuable the feedback can be'. In other words, a well-defined schema is an important condition for producing valuable feedback, with or without an ITS. We expect that in many cases rules of thumb, and sometimes procedures or even algorithms, can be developed for certain types of models. In case of a procedure or an algorithm, which are preferable, feedback is possible on the steps taken as well as on the intermediary models and final model. Otherwise, rules of thumb are often possible. We expect that rules of thumb in the form of phase-by-phase prescriptions, i.e. what a student has to do and in what order, are promising. Although feedback on the steps taken within a phase cannot often be analyzed automatically, we expect nonetheless that the intermediate models can often be analyzed by using model tracing and CBM techniques.

An example is, although in the domain of program design instead of modeling, the design recipe Felleisen et al. use in their book *How To Design Programs* (2001). This recipe for developing functions consists of seven phases.

1. Define the signatures of a function name, the number and types of input parameters, and the type of the output parameter.
2. Describe the purpose of the function shortly.
3. Define some examples as test cases using the signature.
4. Define a stub.
5. Develop a template for the function body.
6. Define the function body using the template and the examples.
7. Test the function.

Such a recipe provides some guidance for a process that can often appear to be overwhelming for a student. For the teacher, intermediary products are visible, which can be inspected so that immediate feedback can be given to the student. Currently, we are developing a phase-by-phase description for modeling thread based programs.

Development of technology. The prototypes described in the chapters 3, 4, and 5 can be further developed.

As is described in chapter 3, the technology developed in the IDEAS project for generating feedback on well-defined tasks is in an advanced stage. This technology can be applied in other domains than mathematics and programming, which will probably bring new problems to solve.

The schema-analysis technique described in chapter 4 is in a prototype stage and can be applied in real author environments. Furthermore, the technique can be used to define constraints on other artifacts such as UML class diagrams.

The framework for supporting different modeling languages can be further developed. The UML class diagram notation can be implemented including structures such as generalization, specialization, inheritance, etcetera, with corresponding functions for analyzing meta-model errors and semantic errors. Next, a second language for modeling structural aspects, as for example ER-notation, can be implemented. The interesting question is then, to which extent the central model can support a model expressed in these two languages.

Bibliography

[1] W. van der Aalst and C. Stahl. *Business Process Modeling, A Petri Net-oriented Approach.* MIT Press, 2011. pages 1, 12

[2] R.J. Abbott. Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894, November 1983. pages 16

[3] J.R. Anderson. *Rules of mind.* Lawrence Erlbaaum Association, Hillsdale, NJ, 1993. pages 47

[4] J.R. Anderson, C.F. Boyle, R. Farrell, and B.J. Reisser. Cognitive principles in the design of computer tutors. In P. Moris, editor, *Modeling cognition*, pages 93–134. Wiley, New York, 1987. pages 35, 47

[5] J.R. Anderson, A.T. Corbett, K.R. Koedinger, and R. Pelletier. Cognitive Tutors: Lessons learned. *The journal of learning sciences*, 4(2):167–207, 1995. pages 35, 47

[6] L. Aroyo and D. Dicheva. Courseware Authoring Tasks Ontology. In *Proceedings of the International Conference on Computers in Education*, ICCE '02, pages 13–19, Washington, DC, USA, 2002. IEEE Computer Society. pages 57

[7] L. Aroyo and D. Dicheva. Authoring support in concept-based web information systems for educational applications. *J. Cont. Engineering Education and Lifelong Learning*, 14(3), 2004. pages 96

[8] L. Aroyo and D. Dicheva. The new challenges for e-learning: The educational semantic web. *Educational technology and Society*, 7(4):59–69, 2004. pages 96

[9] L. Aroyo and R. Mizoguchi. Towards Evolutional Authoring Support Systems. *Journal of interactive learning research*, 15(4):365–387, 2004. pages 84, 96

[10] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge university press, 1999. pages 118

[11] F. Backstrom and A. Ivarsson. Meta-Model Guided Error Correction for UML Models. `http://www.ep.liu.se/undergraduate/abstract.xsql?dbid=8746`, 2006. pages 114

[12] N. Baghaei and A. Mitrovic. Evaluating a Collaborative Constraint-based Tutor for UML Class Diagrams. In *AIED*, pages 533–535, 2007. pages 56

[13] N. Baghaei, A. Mitrovic, and W. Irwin. Problem-solving Support in a Constraint-based Tutor for UML Class Diagrams. *Technology, Instruction, Cognition and Learning*, 4, 2006. pages 50, 56

[14] M. Balaban, A. Maraee, and A. Sturm. Management of Correctness Problems in UML Class Diagrams Towards a Pattern-Based Approach. *International Journal of Information System Modeling and Design (IJISMD)*, 1(4):24–47, 2010. pages 115

[15] B. Beckert, U. Keller, and P.H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002. pages 114

[16] M. Beeson. Design Principles of Mathpert: Software to support education in algebra and calculus. *Kajler, N. (ed.) Computer-Human Interaction in Symbolic Computation*, pages 89–115, 1998. pages 48, 64, 68, 77, 81

[17] G.J. Bex, W. Gelade, F. Neven, and S. Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. In *International Conference on World Wide Web*, pages 825–834. ACM, 2008. pages 124

[18] G.J. Bex, W. Martens, W. Gelade, and F. Neven. Simplifying XML Schema: Effortless Handling of Nondeterministic Regular Expressions. In *International Conference on Management of Data*, pages 731–744. ACM, 2009. pages 132

[19] C. Bokhove, A. Heck, and G.J. Koolstra. Intelligent feedback to digital assessments and exercises (in dutch). *Euclides*, pages 70–74, February 2005. pages 76, 81

[20] C. Bokhove, G.J. Koolstra, P. Boon, and A. Heck. Towards an integrated learning environment for mathematics. In E. Milková and P. Prazák, editors, *8th International Conference on Technology in Mathematics Teaching (ICTMT8)*, 2007. pages 76

[21] G. Booch, I. Jacobson, and J. Rumbaugh. The UML specification documents. Santa Clara, CA: Rational Software Corp. See documents at www.rational.com, 1997. pages 1

[22] P. Boradkar. Design as Problem Solving. In R. Frodeman, J. Thompson Klein, and C. Mitcham, editors, *The Oxford Handbook of Interdisciplinarity*. Oxford University Press, Oxford, 2010. pages 10

[23] M. Broyl, M. Crane, A. Dingel, J. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In T. Kuhne, editor, *MoDELS 2006 Workshops*, pages 318–323. Springer-Verlag, 2007. pages 33

[24] A. Bruggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120:87–98, 1993. pages 124

[25] A. Bruggemann-Klein and D. Wood. One-unambiguous Regular Languages. *Information and Computation*, 140:229–253, 1998. pages 123, 124

[26] P. Brusilovsky. Developing adaptive educational hypermedia systems: From design models to authoring tools. In T. Murray, S. Blessing, and S. Ainsworth, editors, *Authoring Tools for Advanced Technology Learning Environment*, pages 377–409. Kluwer Academic Publishers, 2003. pages 97

[27] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983. pages 22, 132

[28] J. Carter. Instructional learner feedback: A literature review with implications for software development. *The Computing Teacher*, pages 53–55, December 1984. pages 9

[29] D. Charsky. From Edutainment to Serious Gaming: A change in the use of game characteristics. *Games and Culture*, 5 (2):177–198, 2010. pages 22

[30] P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:9–36, 1976. pages 1, 99

[31] I. Chudov. Linear system solver (using determinant). `http://www.algebra.com/algebra/homework/coordinate/linear.solver`, 2004. pages 77

[32] A. Cohen, H. Cuypers, E. Barreiro, and H. Sterk. Interactive mathematical documents on the web. *Algebra, Geometry and Software Systems*, pages 289–306, 2003. pages 80

[33] A.M. Cohen, H. Cuypers, D. Jibetean, and M. Spanbroek. Interactive learning and mathematical calculus. In *Mathematical Knowledge Management*, 2005. pages 77

[34] V.B. Cohen. A reexamination of feedback in computer-based instruction: Implications for instructional design. *Educational Technology*, 25(1):33–37, 1985. pages 9

[35] IMS Global Learning Consortium. IMS LD Specification. `"http://www.imsglobal.org/learningdesign/index.html"`, April 2013. pages 84

[36] M.A. Constantino-González and D. Suthers. A Coached Collaborative Learning Environment for Entity-Relationship Modeling. In C. Gauthler, C. Frasson, and K. VanLehn, editors, *Intelligent Tutoring Systems, Proceedings of the 5th International Conference (ITS 2000)*, pages 324–333. Springer, 2000. pages 51

[37] A.T. Corbett and K.R. Koedinger. Intelligent Tutoring Systems. In M. Helander, T.K. Landauer, and P. Prabhu, editors, *Handbook of Human-Computer Interaction*, pages 849–874. Elsevier Science B.V., 1997. pages 22

[38] A. Cristea. Authoring of Adaptive Hypermedia: Adaptive Hypermedia and Learning Environments. In S.Y. Chen and G.D. Magoulas, editors, *Advances in Web-based Education: Personalized Learning Environments*. IDEA Publishing group, 2004. pages 97

[39] B. Davey and H. Priestly. *Introduction to lattices and order, 2e edition.* Cambridge University Press, 2002. pages 85, 91

[40] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 243–320. North-Holland, Amsterdam, 1990. pages 67

[41] M. Doorman, P. Drijvers, P. Boon, S. van Gisbergen, and K. Gravemeijer. Design and implementation of a computer supported learning environment for mathematics. In *Earli 2009 SIG20 invited Symposium Issues in designing and implementing computer supported inquiry learning environments*, 2009. pages 80

[42] T. Duffy and D. Cunningham. Constructivism: Implications for the design and delivery of instruction. In D.H. Jonassen, editor, *Handbook of research for educational communications and technology*. MacMillian Library Reference, New York, USA, 1996. pages 22

[43] K. Dunbar. Problem Solving. *A Companion to Cognitive Science*, pages 289–298, 1998. pages 28, 34, 40, 45

[44] A. Egyed. Semantic abstraction rules for class diagrams. In *The fifteenth IEEE International Conference on Automated Software Engineering*, 2000. pages 114

[45] I. Erev, A. Luria, and A. Erev. On the effect of immediate feedback. `http://goo.gl/eodze`, 2006. pages 9

[46] G. Fischer, K. Nakakoji, J. Ostwald, G. Stahl, and T. Sumner. Embedding critics in design environments. In Mark T. Maybury and Wolfgang Wahlster, editors, *Readings in intelligent user interfaces*, pages 537–561. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. pages 55

[47] P. Fradet, D. Le Metayer, and D. Perin. Consistency checking for multiple view software architectures. In *Proc. of the Joint European Software Engineering Conference and Symp. on Foundations of Software Engineering, ESEC/FSE'99, Software Engineering Notes 24 (6) or LNCS Vol 1687*, pages 410–428, 1999. pages 114

[48] R.M. Gagne. *The Conditions of Learning and Theory of Instruction.* New York: CBS College Publishing, 1985. pages 10

[49] G. Genova, M. Valiente, and M. Marrero. On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering. *Journal of Object Technology*, 8:107–127, 2009. pages 7, 11

[50] A. Gerdes. *Ask-Elle: a Haskell Tutor.* Gildeprint drukkerijen BV, Enschede, 2012. Phd thesis. pages 80

[51] M.L. Gick. Problem solving strategies. *Educational psychologist*, 2 / (1 and 2):99–120, 1986. pages 34, 45

[52] M.L. Gick and Holyak. Schema induction and analogical transfer. *Coginitive Psychology*, 15:1–38, 1983. pages 33, 44

[53] M. Gogolla and M. Richters. Transformation Rules for UML Class Diagrams. In J. Bézivin and A. Muller, editors, *UML*, pages 92–106. Springer, 1999. pages 114

[54] G. Goguadze, A.G. Palomo, and E. Melis. Interactivity of Exercises in Activemath. In *Proceedings of the 2005 conference on Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences: Sharing Good Practices of Research, Experimentation and Innovation*, pages 109–115, Amsterdam, The Netherlands, 2005. IOS Press. pages 35

[55] S. Gross, X. Zhu, B. Hammer, and N. Pinkwart. Cluster-based feedback provision strategies in intelligent tutoring systems. In *Proceedings of the 11th international conference on Intelligent Tutoring Systems*, ITS'12, pages 699–700, Berlin, Heidelberg, 2012. Springer-Verlag. pages 52

[56] G. Guri-Rosenblit. Distance Education and E-Learning: Not the Same Thing. *Higher Education*, 49 (4):467–493, 2005. pages 2, 21

[57] L. Hall and A. Gordon. A virtual learning environment for entity relationship modeling. *SIGCSE Bull.*, 30(1):345–349, March 1998. pages 52

[58] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77 (1):81–112, 2007. pages 10, 49

[59] A. Heck and L. van Gastel. Diagnostic testing with Maple T.A. In *Electronic Library of Mathematics of the European Mathematical Society*, pages 37–52. Oy WebALT Inc, 2006. pages 65, 77

[60] B. Heeren and J. Jeuring. Canonical Forms in Interactive Exercise Assistants. In J. Carette, L. Dixon, C. Sacerdoti, and S. Watt, editors, *Proceedings Calculemus/Mathematical Knowledge Management, LNAI 5625*, pages 325–340. Springer, 2009. pages 79

[61] B. Heeren and J. Jeuring. Adapting mathematical domain reasoners. In *Proceedings MKM 2010, the 9th International Conference on*

Mathematical Knowledge Management, LNCS 6167, pages 315–330. Springer, 2010. pages 80

[62] B. Heeren and J. Jeuring. Interleaving strategies. In J.H. Davenport et al, editor, *Proceedings of Calculemus/MKM 2011, LNAI 6824*, pages 196–211. Springer, 2011. pages 80

[63] B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010. pages 53, 78

[64] M.J. Hicks. *Problem solving in business and management: hard, soft and creative approaches.* Chapman and Hall, 1991. pages 29, 40

[65] H. Highland. A taxonomy of models. *SIGSIM Simuletter. Dig.*, 4(2):10–17, January 1973. pages 5, 6

[66] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979. pages 1, 14, 124

[67] G. Hoydalsvik and G. Sindre. On the purpose of Object-Oriented Analysis. In *OOPSLA Proceedings*, 1993. pages 8

[68] M. Jackson. *Problem Frames. Analyzing and structuring software development problems.* Addison-Wesley, 2001. pages 7

[69] R. Jeffries, A. Turner, P. Polson, and M. Atwood. The processes involved in designing software. In J.R. Anderson, editor, *Cognitive skills and there acquisition*, pages 255–283. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., 1981. pages 45

[70] J. Jeuring and W. Pasman. Strategy Feedback in an E-Learning Tool for Mathematical Exercises. Technical Report UU-CS-2007-007, Department of Information and Computing Sciences, Utrecht University, 2007. pages 37, 78

[71] J. Jeuring, H. Passier, and S. Stuurman. A Generic Framework for Developing Exercise Assistants. In *In Proceedings of the 8th International Conference on Information Technology Based Higher Education and Training, ITHET*, 2007. pages 81

[72] J. Jeuring and D. Swierstra. Constructing functional programs for grammar analysis problems. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming*

Languages and Computer Architecture, pages 259–269, 1995. pages 88, 91

[73] J. Jeuring and D. Swierstra. *Grammars and Parsing.* Universiteit Utrecht, 2001. Lecture notes - Not published. pages 121

[74] L. Jin, W. Chen, Y. Hayashi, M. Ikeda, R. Mizoguchi, Y. Takaoka, and M. Ohta. An ontology-aware authoring tool - Functional structure and guidance generation. In *Proc. of AI-ED 99, Le Mans*, pages 85–92. IOS Press, 1999. pages 57, 96

[75] D.W. Johnson and R.T. Johnson. Cooperative learning and feedback in technology-based instruction. *In J.V. Dempsey and G.C. Sales (Eds.), Interactive instruction and feedback*, pages 133–157, 1993. pages 9

[76] D.H. Jonassen. Toward a meta-theory of problem solving. *Educational Technology: Research and Development*, 48(4):63–85, 2000. pages 29, 32, 40

[77] S.P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* `http://haskell.org/`, September 2002. pages 58

[78] H. Kaindl. Difficulties in the transition from OO analysis to design. *IEEE Software*, pages 94–102, September 1999. pages 7, 8

[79] S. Kent. Model Driven Engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2002. pages 11

[80] K. Koedinger and J. Anderson. Intelligent Tutoring Goes To School in the Big City. In *AIED*, pages 30–43, 1997. pages 22

[81] A. Kok, H. Pootjes, and M. Sint. Object oriented analysis and design. Open universiteit Nederland, 2009. Lecture notes (in Dutch). pages 1, 3, 16

[82] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995. pages 6

[83] R.W. Kulhavy and W. Wager. Feedback in programmed instruction: Historical context and implications for practice. In J.V. Dempsey and G.C. Sales, editors, *Interactive instruction and feedback*, pages 3–20. Englewood Cliffs, New Jersey, 1993. pages 49

[84] L. Lamport. The future of computing: logic or biology, 2003. Text of a talk given at Christian Albrechts University, Kiel. pages 117, 132

[85] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009. pages 1, 4, 8, 12, 13, 16, 46, 99

[86] N.T Le, F. Loll, and N. Pinkwart. Operationalizing the Continuum between Well-defined and Ill-defined Problems for Educational Technology. *IEEE Transactions on Learning Technologies*, 99:1, 2013. pages 135

[87] J. Lodder, J. Jeuring, and H. Passier. An interactive tool for manipulating logical formulae. In *In M. Manzano, B. Perez Lancho and A. Gil, editors, proceedings of the Second International Congress on Tools for Teaching Logic*, 2006. pages 78, 81

[88] C. Lynch, K. Ashley, V. Aleven, and N. Pinkwart. Defining Ill-Defined Domains: A literature survey. In *Intelligent Tutoring Systems (ITS 2006): Workshop on Intelligent Tutoring Systems for Ill-Defined Domains*, 2006. pages 22, 29, 40, 43, 50, 51, 54, 113

[89] S. Maoz, J.O. Ringert, and B. Rumpe. CDDiff: semantic differencing for class diagrams. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 230–254, Berlin, Heidelberg, 2011. Springer-Verlag. pages 115

[90] M. Marvrikis, A. Macioncia, and J. Lee. Wallis: a web-based ILE for science and engineering students studying mathematics. Electronic Library of Mathematics of the European Mathematical Society, 2006. `http://www.emis.de/proceedings`. pages 77

[91] R. Mayer. *Thinking, problem solving, cognition.* New York: Freeman, 1983. pages 44

[92] J. McKendree. Effective feedback content for tutoring complex skills. *Human Computer Interaction*, 5(4):381–413, December 1990. pages 81

[93] E. Melis and J. Siekmann. Activemath: An intelligent tutoring system for mathematics. In *Seventh International Conference 'Artificial Intelligence and Soft Computing' (ICAISC), volume 3070 of LNAI*, pages 91–101. Springer, 2004. pages 80

[94] J. Mendel. A taxonomy of models used in the design process. *interactions*, 19(1):81–85, January 2012. pages 6

[95] T. Mens, K. Czarnecki, and P. van Gorp. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005. pages 8

[96] J.G. Merriënboer and P.A. Kirschner. *Ten Steps to Complex Learning*. Routledge, 2007. pages 10, 34, 45, 46, 47, 132

[97] B. Meyer. *Touch of Class: Learning to program well with objects and contracts*. Springer, 2009. pages 132

[98] N. Milik, M. Marshall, and A. Mitrovic. Teaching Logical Database Design in ERM-Tutor. In M. Ikeda, K. Ashley, and T. Chan, editors, *Proceedings of Intelligent Tutoring Systems 2006*, volume 4053 of *Lecture Notes in Computer Science*, pages 707–709. Springer, Berlin/Heidelberg, 2006. pages 55

[99] K.E. Miloudi, Y.L. Amrani, and A. Ettouhami. An Automated Translation of UML Class Diagrams into a Formal Specification to Detect UML Inconsistencies. In L. Lavazza, L. Fernandez-Sanz, O. Panchenko, and T. Kanstrén, editors, *ICSEA 2011: The Sixth International Conference on Software Engineering Advances*, pages 432–438, 2011. pages 115

[100] S.K. Milton, E. Kazmierczak, and C.D. Keen. On the Study of Data Modelling Languages using Chisholm's Ontology. In H. Kangassalo, T. Welzer, H. Jaakkola, I. Rozman, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases XIII*, pages 19–36, Amsterdam, 2002. IOS Press. pages 108

[101] A. Mitrovic. NORMIT, a Web-enabled tutor for database normalization. In *Proceedings of the International Conference on Computers in Education ICCE*, pages 1276–1280, 2002. pages 54, 55, 115

[102] A. Mitrovic and K. Hausler. An Intelligent SQL Tutor on the Web. In *International Journal of Artificial Intelligence in Education*, pages 37–44, 2000. pages 55, 115

[103] A. Mitrovic, B. Martin, P. Suraweera, K. Zakharov, N. Milik, J. Holland, and N. McGuigan. ASPIRE: An Authoring System and Deployment Environment for Constraint-Based Tutors. *International Journal on Artificial Intelligence in Education*, 19(2):155–188, 2009. pages 57

[104] A. Mitrovic and A. Weerasinghe. Revisiting ill-definedness and the consequences for ITSs. In *Proceeding of the 2009 conference on Artificial Intelligence in Education*, pages 375–382. Press, 2009. pages 29, 42, 43

[105] A. Mitroviv, S. Ohlsson, and D. Barrow. The Effect of Positive Feedback in a Constraint-based Intelligent Tutor System. *Computers and Education*, 60(1):264–272, January 2013. pages 10

[106] R. Mizoguchi, K. Sinitsa, and M. Ikeda. Task Ontology Design for Intelligent Educational/Training systems. Position paper for ITS '96, Workshop on architectures and methods for designing cost-effective and reusable ITSs, Montreal, 1996. pages 22, 37

[107] N. Moha and Y.G. Guéhéneuc. On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs. In *Proceedings of the 6th International ECOOP Workshop on Object-Oriented Reengineering*, 2005. pages 114

[108] N. Moha, Y.G. Guéhéneuc, L. Duchien, and A.F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 2010. pages 114

[109] A. Moller and M. Schwartzback. *An Introduction to XML and Web Technologies*. Addison-Wesley, 2006. 1st edition. pages 3

[110] J. Moore, C. Dickson-Deane, and K. Galyen. E-learning, online learning, and distance learning environments: Are they the same? *Internet and Higher Education*, 14:129–135, 2011. pages 21

[111] S. Moritz. *Generating and Evaluating Object-Oriented Designs in an Intelligent Tutoring System*. Lehigh University, 2008. Phd thesis. pages 50, 51, 53

[112] S. Moritz and G. Blank. Generating and Evaluating Object-Oriented Designs for Instructors and Novice Students. In *9th International Conference on Intelligent Tutoring Systems, Workshop on Ill-Formed Domains*, 2008. pages 52, 54, 55

[113] E.H. Mory. Feedback Research Revisited. In D.H. Jonassen (ed.), Handbook of research for educational communications and technology, 2003. pages 1, 9, 10, 22, 49

[114] L. Motmans. Voorkennis Wiskunde (in dutch). `https://www.uhasselt.be/Documents/UHasselt/brochures/2010/GeheelInhaalWiskunde.pdf`. pages 34

[115] T. Murray. Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 8:222–232, 2003. pages 56

[116] T. Murray. An Overview of Intelligent Tutoring System Authoring Tools: Updated Analysis of the State of the Art. *Authoring tools for advanced technology learning*, pages 493–546, 2003. pages 36, 39, 50, 56, 57, 83, 84, 96

[117] A. Newell and H.A. Simon. *Human problem solving*. Englewood Cliffs, N.J.: Prentice-Hall, 1972. pages 40

[118] S. Ohlsson and A. Mitrovic. Constraint-based knowledge representation for individualized instruction. *Computer Science and Information Systems*, 3:1–22, 2006. pages 54, 115

[119] H. Passier. A framework for feedback in e-learning systems for data modeling. In *Proceedings of the IADIS International conference*. IADIS, 2008. pages 19, 99

[120] H. Passier. Notes on modeling XML content models. Technical report, Faculty of Informatics, Open Universiteit Nederland, 2010. pages 19

[121] H. Passier and B. Heeren. Modeling XML Content Explained. Technical report, Department of Information and Computing Sciences, Utrecht University, 2011. pages 19, 117

[122] H. Passier and B. Heeren. Modeling XML content models explained. In *In proceedings of the MCCSIS*. IADIS, 2011. pages 19

[123] H. Passier and J. Jeuring. Ontology-based feedback generation in design-oriented e-Learning systems. In *Proceedings of the IADIS International conference*, pages 992–996. IADIS, 2004. pages 18, 21

[124] H. Passier and J. Jeuring. Using Schema Analysis for Feedback in Authoring Tools for Learning Environments. In *Artificial Intelligence*

in Education, AIED, volume 125, pages 911–914. IOS Press, 2005. pages 19

[125] H. Passier and J. Jeuring. Using Schema Analysis for Feedback in Authoring Tools for Learning Environments (extended version). In A. Cristea, R. Carro, and F. Garzotto, editors, *Proceedings of the Third International Workshop on Authoring of Adaptive and Adaptable Educational Hypermedia, A3EH*, pages 13–20, 2005. pages 19, 83

[126] H. Passier and J. Jeuring. Feedback in an interactive equation solver. In *Proceedings of the Web Advanced Learning Conference and Exhibition, WebALT 2006*, pages 53–68. Oy WebALT Inc, 2006. pages 19, 61

[127] C.A. Petri. Kummunikation mit Automaten. PhD thesis. Bonn, Institut für instrumentelle Mathematik, 1962. pages 1

[128] G. Polya. *How to solve it.* Penguin Mathematics, 1985. pages 34, 45

[129] P. Rosmalen. *Supporting the tutor in the design and support of adaptive e-learning.* Datawyse, Maastricht, 2008. Phd thesis Open universiteit Nederland. pages 22

[130] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999. pages 13

[131] S. Russell and P. Norvig. *Artificial intelligence. A modern approach.* Prentice Hall International, 1995. pages 37, 85, 106

[132] G.C. Sales. Adapted and adaptive feedback in technology-based instruction. In J.V. Dempsey and G.C. Sales, editors, *Interactive instruction and feedback*, pages 159–175. Englewood Cliffs, New Jersey, 1993. pages 9, 49

[133] K. Satoh, K. Kaneiwa, and T. Uno. Contradiction Finding and Minimal Recovery for UML Class Diagrams. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 277–280, Washington, DC, USA, 2006. IEEE Computer Society. pages 115

[134] J. Schramm, S. Strickroth, N. Le, and N. Pinkwart. Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System. In G. Michael Youngblood and Philip M. McCarthy, editors, *FLAIRS Conference*. AAAI Press, 2012. pages 52

[135] A. Shalloway and J. Trott. *Design patterns explained.* Addison Wesley, 2002. pages 16

[136] M. Sint, A. Herrewijn, A. Kok, and M. Witsiers. Object oriented programming with Java 2. Open universiteit Nederland, 2009. Lecture notes (in Dutch). pages 1, 3

[137] P.L. Smith and T.J. Ragan. Designing instructional feedback for different learning outcomes. In J.V. Dempsey and G.C. Sales, editors, *Interactive instruction and feedback*, pages 75–103. Englewood Cliffs, New Jersey, 1993. pages 10, 33, 35, 36, 44, 49

[138] M.R. Sobel, A.E.K.and Clarkson. Formal Methods Application: An Empirical Tale of Software Development. *IEEE Transactions on Software Engineering*, 28:308–320, 2002. pages 132

[139] Softmath. Algebrator-algebra help software. `http://www.softmath.com`, 2013. pages 77

[140] I. Sommerville. *Software Engineering.* Addison-Wesley, Harlow, England, 9th edition, 2010. pages 5, 6

[141] J.F. Sowa. *Knowledge representation: Logical, philosophical, and computational foundations.* Brooks/Cole, USA, 2000. pages 37

[142] L. Stojanovic, S Staab, and R. Studer. eLearning based on the Semantic Web. In *WebNet2001 - World Conference on the WWW and Internet*, pages 23–27, 2001. pages 96

[143] S. Stoyanov. *Mapping in the educational and training design.* Print partners Ipskamp, Enschede, Nederland, 2001. Phd thesis university of Twente, Enschede. pages 10, 22, 28, 40

[144] M. Striewe and M. Goedicke. Automated checks on UML diagrams. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, ITiCSE '11, pages 38–42, New York, NY, USA, 2011. ACM. pages 51, 55, 113, 115

[145] P. Suraweera and A. Mitrovic. KERMIT: A Constraint-based Tutor for Database Modeling. In *Proc. 6th Int. Conf on Intelligent Tutoring Systems ITS 2002*, pages 377–387, 2002. pages 55, 116

[146] S.D. Swierstra and L. Duponcheel. Deterministic, Error-Correcting Combinator Parsers. In *Advanced Functional Programming*, pages 184–207. Springer-Verlag, 1996. pages 69

[147] B. Tekinerdogan. *Synthesis-based software architecture design.* Print partners Ipskamp, Enschede, Nederland, 2000. Phd thesis university of Twente, Enschede. pages 10

[148] O.D. Temur. Analysis of Prospective Classroom Teachers' Teaching of Mathematical Modeling and Problem Solving. *Eurasia Journal of Mathematics, Science and Technology Education*, 8 (2):83–93, 2012. pages 10

[149] D.C. Tsichritzis and F.H. Lochovsky. *Data models.* Prentice-Hall Software Series, 1982. pages 106

[150] S.A. Tucker. Evaluation as feedback in instructional technology: The role of feedback in program evaluation. *In J.V. Dempsey and G.C. Sales (Eds.), Interactive instruction and feedback*, pages 301–342, 1993. pages 9

[151] J. Ullman and A. Aho. *Foundations of Computer Science.* Out of print, 1992. `http://infolab.stanford.edu/~ullman/focs.html`. pages 10

[152] A Van Gundy. *Techniques of structured problem solving.* New York: Van Nostrand, 1997. pages 29, 42

[153] J. in 't Veld. *Analyse van organisatieproblemen.* Elsevier, 1983. pages 25

[154] W3C. Resource Description Framework (RDF), 2004. pages 37, 84

[155] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA (2nd Edition).* Addison Wesley, 2003. pages 99

[156] D.A. Watt and D.F. Brown. *Programming language processors in Java.* Prentice Hall, 2000. pages 123

[157] L. Wiegerink, J. Bijpost, M. de Groot, D. Oosten, and L. Hozeman. Databases. Open universiteit Nederland, 2009. Lecture notes (in Dutch). pages 1

[158] L. Wiegerink, H. Pootjes, and H. Passier. XML: Theory and applications. Open universiteit Nederland, 2008. Lecture notes (in Dutch). pages 1, 3

[159] J.M. Wing. Weaving Formal Methods into the Undergraduate Computer Science Curriculum. In *Proceedings of AMAST*, pages 2–9, 2000. pages 132

[160] J. van der Woude. Process modeling. Open universiteit Nederland, 2011. Lecture notes (in Dutch). pages 1

[161] K. Zakharov, A. Mitrovic, and S. Ohlsson. Feedback Micro-engineering in EER-Tutor. In *AIED*, pages 718–725. Press, 2005. pages 55

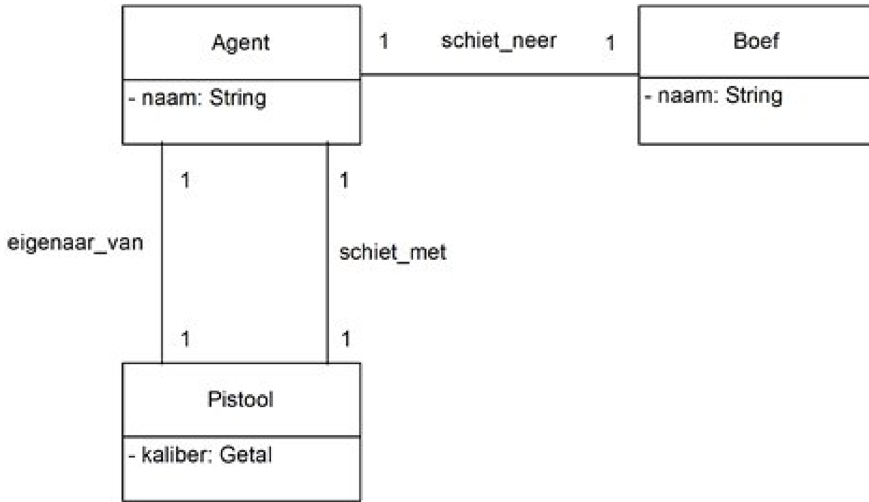
Samenvatting

In deze samenvatting geef ik een overzicht van het onderzoek waaraan ik de afgelopen jaren heb gewerkt. Ik veronderstel daarbij geen specifieke voorkennis. Het onderzoek gaat over het automatisch produceren van feedback tijdens het maken van modellen. Deze modellen worden gebruikt voor het ontwikkelen van computerprogramma's. Ik zal eerst uitleggen wat met 'modellen' en 'feedback' wordt bedoeld. Daarna leg ik in het kort de probleemstelling uit. Hier zal duidelijk worden dat automatische feedback belangrijk is. Vervolgens presenteer ik een raamwerk waarmee we feedback automatisch kunnen produceren. In dit onderzoek is niet dit hele raamwerk ontwikkeld. In plaats daarvan zijn vier aspecten van dit raamwerk onderzocht en uitgewerkt. In de rest van deze samenvatting wordt ingegaan op deze vier aspecten.

Modellen

Laten we beginnen met een voorbeeld van een model. Stel, we willen een computerspel maken. In het spel moet een politieagent een boef vangen. Om ervoor te zorgen dat het spel levensecht wordt, interviewen we een politieagent. In het interviewverslag komt de volgende zin voor: 'De agent schiet de boef met zijn pistool neer'. Hiervan maken we een model dat later wordt gebruikt om het computerprogramma mee te maken. Figuur 8.1 toont het resultaat.

Figuur 8.1 is een klassendiagram. Een klassendiagram wordt gemaakt met behulp van een modelleertaal. In deze taal beschrijven we de wereld in termen van klassen en relaties tussen deze klassen. In figuur 8.1 zien we drie klassen, namelijk Agent, Boef en Pistool. Elke klasse wordt gekarakteriseerd door zijn attributen. Zo hebben een agent en een boef een naam en heeft een pistool een kaliber. Verder zien we drie relaties, namelijk 'schiet_neer', 'eigenaar_van' en 'schiet_met'. Aan beide uiteinden van elke relatie staat



Figuur 8.1: Een voorbeeld model

een getal. Kijken we naar de relatie 'schiet_neer', dan moeten we dit lezen als: 'Elke agent schiet één (de rechter één) boef neer' en 'Elke boef wordt neergeschoten door één (de linker één) agent'. Het model zegt verder dat elke agent eigenaar is van één pistool en dat een agent schiet met zijn pistool[1].

Modellen zijn belangrijk bij het ontwerpen van een computerprogramma. Een computerprogramma bestaat al snel uit honderden, zo niet duizenden regels programmacode en het is erg lastig om daar het overzicht in te houden. In een model kunnen we ons gemakkelijker concentreren op de grote lijnen en kunnen we allerlei details eenvoudig weglaten. Verder kan aan de hand van een model als in figuur 8.1 beter gecommuniceerd worden met bijvoorbeeld een inhoudsdeskundige dan aan de hand van programmacode die over het algemeen moeilijk is te lezen door een niet-programmeur.

Er bestaan verschillende typen modellen. Zo zijn er modellen voor het vastleggen van de structuur van een computerprogramma (figuur 8.1), het gedrag van een computerprogramma en de context waarin een computerprogramma wordt gebruikt. Een andere belangrijke indeling is het onderscheid tussen analyse- en ontwerpmodellen. De eerste worden vooral gebruikt om 'de wereld die we willen vastleggen in een computerprogramma' beter te begrijpen. Modellen van de tweede soort worden gebruikt om het computerprogramma te ontwerpen. Er is een groot aantal modelleertalen, zowel

[1]Om aan te geven dat de agent schiet met *zijn of haar eigen* pistool moeten we nog een extra OCL constraint toevoegen. We laten dit nu buiten beschouwing

grafische talen als talen die meer wiskundig van aard zijn.

Elke taal kent zowel een syntaxis als een semantiek. De syntaxis geeft aan wanneer een taal 'netjes' wordt gebruikt. Zo schrijven we in het Nederlands 'hij loopt' in plaats van 'hij loop'. Ook elke modelleertaal heeft zo een set aan regels die gevolgd moet worden. In het klassendiagram in figuur 8.1 moet elke klasse bijvoorbeeld een naam hebben en moet elke relatie verbonden zijn met twee klassen. De semantiek geeft aan hoe we de taal moeten interpreteren. Zo betekent een klasse in een klassendiagram een verzameling objecten van eenzelfde soort (bijvoorbeeld de klasse Agent geeft aan dat er agenten bestaan met elk een eigen naam).

Doordat elk taalelement een bepaalde betekenis heeft, heeft een model als geheel ook een bepaalde betekenis. We hebben net gezien hoe we het model in figuur 8.1 moeten interpreteren. Duidelijk zal zijn dat een model als in figuur 8.1 meer eenduidig is dan een beschrijving in natuurlijke taal. Bijvoorbeeld de zin 'De agent schiet de boef met zijn pistool neer ' kan op drie manieren worden geïnterpreteerd[2]! Het model in figuur 8.1 representeert één van deze interpretaties.

Hiermee hebben we gelijk een probleem te pakken: Hoe weten we nu of de semantiek van een model juist is? In ons geval is er maar één manier om daar achter te komen en dat is teruggaan naar de agent die we hebben geïnterviewd en vragen of onze interpretatie juist is. Een andere vraag is of het model niet eenvoudiger kan, bijvoorbeeld door het aantal modelelementen te verminderen.

Het maken van een goed model voor een computerprogramma is vaak moeilijk. We kunnen vandaag de dag in veel gevallen niet precies aangeven hoe een goed model voor een computerprogramma moet worden gemaakt. Een voorbeeld daarvan is het maken van een klassendiagram zoals in figuur 8.1. Het maken van zo'n model is vooral een creatieve activiteit. Er zijn ook uitzonderingen. In dit proefschrift wordt daarvan in hoofdstuk 6 een voorbeeld gegeven. De uitkomst is een heel precies stappenplan (ook wel een strategie genoemd) voor één bepaald soort model; zo precies dat als we de eisen weten die we stellen aan het model, het model automatisch gemaakt kan worden en we gegarandeerd over een goed model beschikken.

[2]De drie interpretaties zijn: (1) de agent schiet met zijn eigen pistool de boef neer, (2) de agent schiet de boef neer met het pistool van de boef, en (3) de agent schiet (met een pistool) de boef neer die ook een pistool heeft.

Feedback

Feedback is cruciaal in het onderwijs. Het meeste leer je door te doen en te horen wat goed is gegaan en wat beter kan. Bij complexe vaardigheden, zoals het maken van modellen voor computerprogramma's, is het meestal goed om direct feedback te krijgen en niet pas achteraf. Tot nu toe kan dat eigenlijk alleen maar wanneer een docent meekijkt terwijl een student een model maakt.

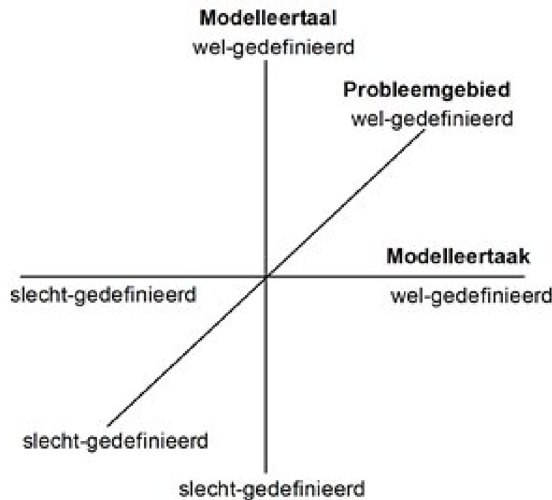
Met feedback kunnen we een proces sturen. Een bekend voorbeeld is de thermostaat van een cv-installatie. Om een feedbacksysteem te laten werken, zijn drie functies nodig. Ten eerste moeten we de *actuele toestand* meten (de temperatuur in de huiskamer). Ten tweede moeten we een *doel* hebben waar naar wordt gestreefd (bijvoorbeeld een temperatuur van 20 graden Celsius). Ten derde moeten we een *regelaar* hebben die de actuele toestand vergelijkt met het doel en maatregelen neemt zodat het doel wordt bereikt (de regelaar laat de cv gedurende een aantal minuten branden als de temperatuur lager is dan de gewenste temperatuur).

In de onderwijssituatie die wij bestuderen, het maken van modellen voor computerprogramma's, wordt de actuele toestand gevormd door het finale model en eventuele tussentijdse modellen die een student oplevert. Het doel is dat de student een goed model maakt. Een model is goed als het zowel syntactisch als semantisch correct is. Dat eerste, de syntaxis, is eenvoudig te controleren. Het tweede, de semantiek, zo hebben we net gezien, is vaak lastig te controleren, temeer omdat veelal meerdere modellen semantisch gezien goed zijn. De regelaar die nodig is om feedback te produceren tijdens het maken van modellen, is veel complexer dan de regelaar in een cv-installatie: het goed of niet goed zijn van een model hangt van veel aspecten af en ook kan een model op verschillende manieren worden gemaakt, waarbij de ene manier handiger is dan de andere.

Figuur 8.2 geeft preciezer aan wanneer we feedback kunnen geven in het geval van modelleeropgaven. Om zinvol feedback te kunnen geven op het uiteindelijke model en de stappen die daartoe zijn gezet, moet aan de volgende drie eisen zijn voldaan:

- De modelleertaal moet wel-gedefinieerd zijn. We hebben in het voorbeeld, waarin de agent de boef neerschiet, gezien dat natuurlijke taal, zoals Nederlands, niet altijd eenduidig is. De taal 'klassendiagram' is wel eenduidig.

- De modelleertaak moet wel-gedefinieerd zijn. Hieronder verstaan we de beginsituatie, het doel dat wordt nagestreefd en de strategie die



Figuur 8.2: Modelleertypen

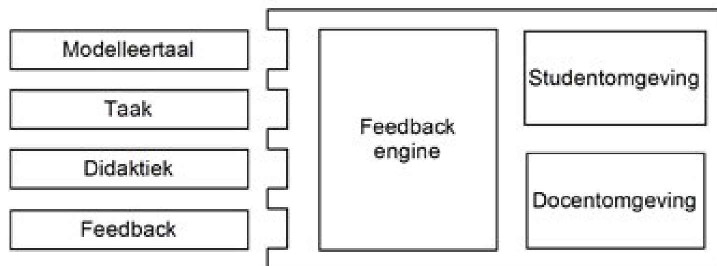
wordt gevolgd. Als bijvoorbeeld het doel niet duidelijk is, kunnen we ook niet aangeven of dit is bereikt. Als de strategie niet is gedefinieerd, kunnen we niet aangeven in hoeverre de aanpak volgens de beoogde strategie is geweest.

- Het probleemgebied moet wel-gedefinieerd zijn. Als het gebied waarvan we een model maken niet wel-gedefinieerd is, is het moeilijk om te bepalen of het model het probleemgebied goed weergeeft. Een voorbeeld hiervan hebben we gezien: de interviewtekst als uitgangspunt voor het maken van het klassendiagram is onvoldoende wel-gedefinieerd om te bepalen of het model semantisch correct is.

Waarom is automatische feedback belangrijk?

De Open Universiteit biedt afstandsonderwijs. Het studiemateriaal bestaat uit leer- en werkboeken met daarin opgaven en uitwerkingen. Er worden colleges gegeven, zowel in een collegezaal als via een elektronische klas. Het aantal colleges is beperkt. Meestal ongeveer vijftien uur per vak verdeeld over vijf tot tien bijeenkomsten. Verder is er bij elke cursus een discussiegroep, waarin studenten vragen kunnen stellen. Medestudenten en docenten reageren op de vragen die gesteld worden.

Probleem is dat het aantal opgaven dat door de student gemaakt moet worden om voldoende vaardig te worden in modelleren groot is. Bij elke



Figuur 8.3: Het raamwerk

opgave wordt een uitwerking gegeven, maar bij veel opgaven zijn meerdere goede antwoorden mogelijk. Een student die een (net iets) ander antwoord heeft, blijft met vragen zitten als: 'Is mijn antwoord wel goed?' en 'Waar ben ik in de fout gegaan?'. Het is ondoenlijk om alle alternatieve uitwerkingen tijdens de colleges of in de discussiegroepen te bespreken.

Een oplossing

De wens is om een elektronische omgeving aan studenten aan te bieden, waarin zij modellen kunnen maken en waarbij automatisch direct feedback wordt gegeven op het eindresultaat en, als het mogelijk is, op de tussenstappen die zijn gezet. Zo'n elektronische leeromgeving die intelligente feedback geeft, wordt ook wel een Intelligent Tutor Systeem (ITS) genoemd. Er bestaan al een aantal ITSs, bijvoorbeeld voor het maken van wiskunde- en natuurkundeopgaven. Er zijn weinig ITSs voor het maken van modellen voor computerprogramma's. Kenmerkend voor al deze ITSs is dat per opgave, in ons geval per model, alle fouten in het ITS moeten worden ingevoerd die door een student kunnen worden gemaakt. Vervolgens moet per fout bijpassende feedback worden opgenomen. In het geval van modelleeropgaven is dit een haast onmogelijke zaak. Dit wordt nog eens verergerd doordat er meestal meerdere goede modellen mogelijk zijn.

Om hier een oplossing voor te bieden, stellen we het volgende raamwerk voor (zie ook figuur 2.3 in hoofdstuk 2):

Het raamwerk bevat een omgeving waarin een student modelleeropgaven kan maken. Er wordt feedback gegeven op het eindresultaat en zo mogelijk op tussenliggende resultaten en de strategie die is gevolgd. Het raamwerk biedt ook een omgeving voor de docent, waarin hij/zij modelleeropgaven kan ontwerpen en invoeren, inclusief de antwoorden en feedback.

In plaats van dat de docent *per opgave* alle mogelijk goede antwoorden en fouten met bijbehorende feedback invoert, voert hij/zij nu zoveel mogelijk algemene beschrijvingen in, ook wel ontologieën genoemd, die horen bij een bepaalde *klasse van modellen*. Een voorbeeld is de klasse 'klassendiagrammen'. Met deze ontologieën kunnen we automatisch feedback produceren. De volgende ontologieën onderkennen we:

- *Modelleertaal* – Deze ontologie beschrijft de modelleertaal die wordt gebruikt en wanneer deze netjes wordt gebruikt. Een voorbeeld is de taal voor klassendiagrammen.

- *Taak* – De taakontologie beschrijft in algemene termen de opdracht en de strategie waarmee de opdracht kan worden uitgevoerd. Voor sommige taken beschikken we over een heel precieze strategie, voor andere is deze strategie in de vorm van vuistregels. Verder beschrijft de taak ook het doel dat moet worden bereikt.

- *Didactiek* – Deze ontologie beschrijft allerlei didactische zaken, zoals de stapgrootte die wordt toegelaten in relatie tot de strategie, of de student het model moet maken volgens één bepaalde strategie of dat hij/zij zelf een strategie kan kiezen, en welke vorm van feedback wordt gebruikt.

- *Feedback* – Deze ontologie beschrijft de verschillende soorten feedback die we kunnen onderscheiden, zoals directe feedback, uitgestelde feedback en positieve feedback.

De gedachte is dat de ontologieën verwisselbaar zijn. Voor een andere modelleertaal hoeven we alleen de ontologie Modelleertaal te wisselen, enzovoorts. De feedback engine (de regelaar) observeert het model dat de student aan het maken is en produceert op basis van de ontologieën feedback. In het geval de ontologieën heel precies zijn, oftewel wel-gedefinieerd, kan de feedback ook heel precies zijn. Belangrijk is dus dat we over deze precieze ontologieën beschikken.

Bij het ontwerpen van cursusmateriaal en opgaven spelen veel modelleeraspecten een rol. Het ligt dus voor de hand om ook de docent te voorzien van feedback wanneer hij/zij cursusmateriaal en/of opgaven aan het ontwikkelen is.

Vier onderzoeksvragen

In dit onderzoek is niet het hele raamwerk ontwikkeld en geïmplementeerd. In plaats daarvan zijn vier aspecten bekeken die in afzonderlijke hoofdstukken zijn beschreven. De onderzoeksvragen zijn de volgende:

1. Hoe kunnen we automatisch feedback produceren voor een wel-gedefinieerde taak, een wel-gedefinieerde taal en een wel-gedefinieerd probleemgebied?
2. Hoe kunnen we een docent ondersteunen bij het ontwikkelen van bijvoorbeeld taakontologieën, met als randvoorwaarde dat het ontwikkelproces flexibel blijft?
3. Hoe kunnen we meerdere modelleertalen ondersteunen?
4. Wat moet er gebeuren om van een slecht gedefinieerde taak een goed gedefinieerde taak te maken?

In de volgende paragrafen gaan we kort op elk van deze vragen in.

1. Het oplossen van lineaire vergelijkingen

Als wel-gedefinieerde taak is in hoofdstuk 3 gekozen voor het oplossen van een stelsel lineaire vergelijkingen. Waarschijnlijk kent u dit nog uit uw middelbare schoolperiode. Een voorbeeld hiervan is:

$$\begin{cases} y & = x - 1 \\ \frac{1}{2} \cdot x & = 2 - y \end{cases}$$

Via een aantal stappen moet dit stelsel worden herschreven naar een oplossing in de volgende vorm:

$$\begin{cases} y & = 1 \\ x & = 2 \end{cases}$$

Het oplossen van een stelsel lineaire vergelijkingen is een vorm van modeltransformeren en dit is een vorm van modelleren.

De taal (de wiskunde taal van lineaire vergelijkingen) is wel-gedefinieerd. Elk stelsel vergelijkingen bestaat uit een aantal vergelijkingen, waarbij elke vergelijking weer bestaat uit een linker-expressie, gevolgd door een =-teken, gevolgd door een rechter-expressie. Elke expressie is volgens een vast aantal regels opgebouwd uit getallen, variabelen en rekenkundige operatoren ($+$, $-$, $\times$ of $/$).

Voor elk van deze niveaus kunnen we herschrijfregels definiëren. Twee voorbeelden zijn:

- Voor een expressie geldt dat we twee getallen $(3 + 5)$ bij elkaar op kunnen tellen tot een getal (8).
- Op het niveau van een vergelijking geldt dat we links en rechts eenzelfde getal mogen optellen of aftrekken. Dus $x + 2 = 5$ kunnen we herschrijven tot $x = 3$.

Voor het oplossen van een stelsel lineaire vergelijkingen kunnen we alle regels in kaart brengen. Verder kunnen we ook de strategie heel precies definiëren. Bekende oplosmethoden zijn de substitutiemethode en de combinatiemethode.

Met behulp hiervan zijn we in staat om zowel de stappen die een student uitvoert te volgen alsook het eindresultaat te beoordelen op correctheid. Verder zijn we in staat om heel gedetailleerd feedback te geven, zowel op stapniveau als over de correctheid van de oplossing. Een ander groot voordeel is dat we niet alleen feedback geven op slechts één opgave, maar op alle opgaven die deel uitmaken van de klasse 'het oplossen van lineaire vergelijkingen'.

2. Schema-analyse

Het ontwerpen en implementeren van cursusmateriaal in een elektronische leeromgeving is ingewikkeld, tijdrovend en foutgevoelig. Om het aantal fouten te verminderen en om een docent te helpen, zouden we gebruik kunnen maken van templates. Via zo'n template wordt een docent gedwongen om bepaalde velden in te vullen en hierbij een bepaalde volgorde aan te houden. Nadeel van deze aanpak is dat het weinig flexibel is: een docent kan alleen maar het template volgen en dus zijn alternatieve ontwerpen en implementaties niet mogelijk. Verder moet bij een nieuwe vorm van onderwijs het template flink worden aangepast.

In hoofdstuk 4 introduceren we schema-analyse als techniek om cursusmateriaal te controleren op bepaalde eigenschappen. Als voorbeeld zijn een aantal functies geïmplementeerd die controleren of begrippen die in het cursusmateriaal worden gebruikt aan bepaalde eisen voldoen. Deze eisen zijn: compleetheid (worden alle begrippen die worden gebruikt wel ergens gedefinieerd), tijdigheid (worden de begrippen wel op tijd gedefinieerd), recursiviteit (zijn er begrippen die in termen van zichzelf zijn gedefinieerd), correctheid (zijn de termen de worden gebruikt in de cursus correct ten opzichte van een domeinontologie), synoniemen (zijn er begrippen met verschillende namen, maar met eenzelfde definitie) en homoniemen (zijn er begrippen met eenzelfde naam, maar met verschillende definities).

Het voordeel is dat een docent nu zijn/haar eigen cursusmateriaal kan ontwikkelen en dat deze functies het uiteindelijke resultaat controleren op algemene eigenschappen. Verder zijn deze functies eenvoudig aan te passen, aan en uit te zetten en aan te vullen met andere functies.

3. Het ondersteunen van meerdere modelleertalen

Studenten moeten meestal meerdere modelleertalen leren, bijvoorbeeld een grafische (zoals de taal voor klassendiagrammen) en een meer wiskundige taal. Nu betekent het veranderen van modelleertaal het volledig opnieuw implementeren van alle opgaven in een ITS, terwijl bijvoorbeeld de taak hetzelfde blijft.

In hoofdstuk 5 introduceren we een raamwerk waarin meerdere modelleertalen worden ondersteund en beschrijven we aan welke eisen moet worden voldaan als dit raamwerk wordt gebruikt. In het raamwerk wordt onderscheid gemaakt tussen syntaxisfouten, semantiekfouten en metamodelfouten. De eerste twee zijn eerder besproken. Een voorbeeld van een metamodelfout is inconsistentie, bijvoorbeeld als een model enerzijds aangeeft dat een auto drie wielen heeft en anderzijds minstens vier.

Het raamwerk gaat ervan uit dat we per taak één referentiemodel als modeluitwerking hebben, dat door de docent wordt gemaakt. Dit referentiemodel is gemaakt in een taal die genoeg uitdrukkingskracht heeft om alle talen waarin studenten moeten modelleren te kunnen ondersteunen. Verder zijn er functies die een model, gemaakt door een student, vertalen naar de taal waarin het referentiemodel is gemaakt. Na deze vertaling kan het model, gemaakt door de student, worden vergeleken met het referentiemodel, waarmee de semantiek wordt gecontroleerd en worden geanalyseerd op metamodelfouten. Voordelen zijn dat er nu slechts één set aan analysefuncties nodig is en dat er per taak maar één referentiemodel nodig is. Nadeel is dat als er een modelleertaal bijkomt waarmee studenten moeten oefenen, er voor deze taal eigen vertaalfuncties moeten worden toegevoegd die het model, gemaakt door een student, vertalen naar de taal waarin het referentiemodel is gemaakt.

Van het raamwerk is alleen een eerste prototype geïmplementeerd, namelijk het klassendiagram.

4. Van slecht - naar goed gedefinieerde taken

In hoofdstuk 6 laten we zien wat het betekent om van een slecht-gedefinieerde taak een goed-gedefinieerde taak te maken. Als voorbeeld is het modelleren

van XML content modellen genomen. We gaan hier verder niet in op wat voor modellen dit zijn.

Bekend was al dat dit soort modellen beschreven worden in een wiskundige taal, namelijk de reguliere-expressie taal. In bijna alle boeken over XML wordt hiervan geen gebruik gemaakt. Ook wordt in de boeken over XML geen strategie gegeven om deze modellen te maken. Studenten moeten dus op creatieve wijze deze modellen maken. De praktijk heeft uitgewezen dat nogal wat van deze modellen semantisch niet correct zijn.

Omdat XML content modellen worden uitgedrukt in de reguliere-expressie taal, is de modelleertaal wel-gedefinieerd. Vervolgens zijn alle nodige regels geïnventariseerd om zo'n content model te herschrijven naar een bepaalde eindvorm (het doel waarnaar we streven). Daarnaast worden er twee eindvormen onderkend en is er voor elk van deze eindvormen een eigen strategie ontwikkeld.

Hiermee is het modelleren van XML content modellen een voorbeeld dat in figuur 8.2 is verschoven langs de as Modelleertaak van slecht- naar wel-gedefinieerd. Dit heeft twee belangrijke voordelen. Ten eerste kunnen we nu gedetailleerd feedback geven, zowel op het uiteindelijke resultaat als ook op de tussenstappen. Ten tweede kunnen we studenten leren hoe op een efficiënte en effectieve manier dit soort modellen gemaakt kunnen worden.

Dankwoord

Als eerste dank ik mijn promotor Lex Bijlsma. Ik ben zeer vereerd onder zijn begeleiding dit proefschrift te hebben mogen schrijven. Het was Lex die aangaf dat ik een proefschrift kon schrijven op basis van de artikelen die ik de afgelopen jaren heb geschreven. Hiermee was het mogelijk om binnen afzienbare tijd dit proefschrift af te ronden.

Ik dank Johan Jeuring en Bastiaan Heeren. Zij zijn mede-auteur van enkele artikelen. Beiden hebben mij geïntroduceerd in het Informatica-onderzoek en hebben mij veel geleerd, onder andere op het gebied van functioneel programmeren en het schrijven van artikelen.

Een aantal mensen heeft op dit proefschrift of delen hiervan commentaar gegeven. Sylvia Stuurman en Lex Bijlsma hebben het hele proefschrift minutieus doorgenomen, Lloyd Rutledge heeft hoofdstuk één becommentarieerd, en Bert Eldering, Nynke Kooistra en Diede Passier hebben de Nederlandstalige samenvatting gelezen. De artikelen die ten grondslag liggen aan dit proefschrift zijn becommentarieerd door Lex Bijlsma, Bastiaan Heeren, Johan Jeuring, Josje Lodder en Frans Mofers. Mijn dank hiervoor. Ook dank ik de leden van de promotiecommisie, te weten Erik Barendsen, Marko van Eekelen, Bastiaan Heeren, Stef Joosten en Ruurd Kuiper. Zij hebben in korte tijd het proefschrift gelezen en waardevolle commentaren gegeven.

Ik dank Harold Pootjes voor de vele inspirerende discussies die we de afgelopen jaren hebben gevoerd. Naast dat deze discussies invloed hebben gehad op de inhoud van dit proefschrift, hebben ze ook tot een aantal leesgroepen binnen ons domein Softwaretechnologie geleid en vormen ze mede een basis voor het vervolgonderzoek waar we nu aan werken (Vakdidactiek Informatica). Speciale herinneringen heb ik daarbij aan onze weken in Frankrijk samen met Paula en Diede.

Dit proefschrift is geschreven in een periode dat Diede ziek was, veel in het ziekenhuis heeft gelegen en zorg nodig had. Zonder iedereen bij naam te noemen, realiseer ik me dat het schrijven van dit proefschrift en tegelijkertijd te zorgen voor Diede mogelijk was door de hulp van collegae, familie en

vrienden. Mijn dank hiervoor.

Frank Wester dank ik voor de manier waarop hij een oogje in het zeil hield en mij de ruimte gaf om dit proefschrift te schrijven. Ik voelde me hierdoor gesteund. Annnemiek Herrewijn en Arjan Kok dank ik omdat ze steeds klaar stonden om bijeenkomsten over te nemen en dat ook een aantal keren hebben gedaan.

Tenslotte dank ik Sylvia Stuurman en Eric Mollee omdat zij mijn paranimfen willen zijn.

Curriculum Vitae

Harrie Passier

18 augustus 1962	Geboren te Hilversum
1975 - 1980	MAVO-4, St. Aloysiusschool te Hilversum
1980 - 1984	Opleiding A-Verpleegkunde, Streekziekenhuis Gooi-Noord
1984 - 1986	Militaire dienst
1984 - 1987	VWO exacte vakken, PBNA
1986 - 1987	Opleiding CCU-verpleegkunde, St. Antonius ziekenhuis te Nieuwegein
1987 - 1989	CCU-Verpleegkundige, St. Antonius Ziekenhuis te Nieuwegein
1987 - 1992	HTS Elektrotechniek/Technische Computerkunde (cum laude), Hogeschool Utrecht
1989 - 1995	Projectleider, KPN Telecom te Utrecht
1993 - 2002	WO Informatica, Open Universiteit
1995 - 1999	Wetenschappelijk medewerker, projectleider en werkveld-coördinator, KPN Research te Leidschendam
1999 - 2002	Projectmanager en consultant, KPN Softwarehuis te Groningen
2001 - 2002	Docent Human Computer Interaction, Hanzehogeschool te Groningen
2002 - heden	Universitair docent Informatica, Open Universiteit