Considering Ill-Definedness Of Problem Tasks Under The Aspect Of Solution Space

Nguyen-Thinh Le\textsuperscript{a}, Niels Pinkwart\textsuperscript{b}
\textsuperscript{a}Department of Informatics, University of Hamburg, Germany
\textsuperscript{b}Department of Informatics, Clausthal University of Technology, Germany
le@informatik.uni-hamburg.de
niels.pinkwart@tu-clausthal.de

Abstract: Recently, some researchers on intelligent tutoring systems (ITS) have been moving their focus to ill-defined domains, trying to develop modeling and tutoring approaches for these domains. Several workshops on this issue have been organized [1, 2]. However, up to now a generally accepted definition of ill-definedness has not emerged yet. Several characteristics, which typical ill-defined domains should have, have been proposed in [12]. This set of characteristics should not be viewed as a check list for characterizing "what is ill-defined" because many domains exist on the continuum between well-definedness and ill-definedness. This unclear distinction leads to difficulties at choosing an appropriate modeling approach for a learning domain. In this paper, we propose to classify instructional task into the following four classes with respect to the size of their solution space: 1) tasks which have one single solution, 2) tasks which can be solved by one solution strategy and implemented by different variants, 3) tasks which can be solved by alternative solution strategies and implemented by different variants, and 4) tasks whose solutions cannot be verified and their space is open-ended. The goal of this paper is to help ITS developers choose between model-tracing and constraint-based modeling approaches based on the class that their instructional task belongs to.

Keywords: ill-definedness, intelligent tutoring systems, constraint-based, model-tracing.

Introduction

Recently, research on ITSs has started considering ill-defined domains. A domain is meant an area of study or a set of problem tasks for practicing. Current research aims at identifying characteristics which make a domain ill-defined, and developing appropriate approaches to deal with them. However, up to now a generally accepted definition of ill-definedness has not emerged yet. An example: at a recent workshop on ill-defined domains, there was a presentation about an intelligent tutoring system for database design [15]. Actually, this is an ill-defined domain because researchers agree that design is an ill-defined activity. However, several attendees at the workshop countered that the problem tasks provided by this system do not form an ill-defined domain because the problem statements are so over-specified that the student is left to map given information in the problem statements into constructs of the database (entities and relationships) in a rather well-defined manner. Similar to this system, many other systems have been claimed to tutor in ill-defined domains. However, their problem tasks can be solved by a single solution strategy. The techniques these systems use to limit student’s options (and make the task well-defined) are: 1) to make the problem statement more detailed, or 2) to design the user interface so that it restricts the solver by predefining certain structures for solutions.
In this paper, we classify problem tasks with respect to the size of the solution space. Our aim is to identify the limits of the two most widely used modeling approaches for ITS systems: model-tracing tutors and constraint-based modeling. In the next section we define the term solution strategy and demonstrate that a problem task, especially in ill-defined domains, often can be solved by different solution strategies. In the second section, we classify problem tasks into five classes. In the third section, we analyze which classes of problems modeling approaches (like model-tracing or constraint-based) can be applied to.

1. Two Variability Levels of Solutions

1.1 Solution Strategy

A problem task can be solved by different strategies. We demonstrate this in three domains: arithmetic, programming and UML design. While arithmetic is often considered a well-defined domain because solutions can be verified as correct or incorrect, the task of creating UML diagrams is generally accepted as a design task because there is no way to verify the correctness of a solution (an UML diagram), rather a solution should be viewed as acceptable if it fulfills the requirements given in a problem statement which is typically given in text format. The domain of programming lies between the two extremes of well-definedness and ill-definedness because the correctness of a programming solution (a program) can be checked against a test bed (which normally can be specified) but the task of programming can also be considered a design task because a programmer has to decide for an appropriate solution strategy and to select appropriate programming constructs.

In the domain of arithmetic, for example, many people multiply two numbers using the method of decimal offsets. There are, of course, other ways to carry out such a computation, e.g. the Russian peasant algorithm. It reduces multiplication to four elementary operations: doubling a number, dividing a number by two, subtracting one, and adding two numbers. Even after the basic algorithmic idea has been chosen, there are still different ways to implement it (e.g., a recursive or an iterative solution).

Analog to arithmetic, alternative solution strategies can also be identified in the domain of programming. A simple example problem is the Investment problem which requests the student to write a program to compute the return after investing an amount of money at a constant yearly interest rate. To solve that problem, the following strategies can be applied by using pure Prolog¹, which is a programming language for logic programming:

1. Analytic strategy: the profit of investing a sum of money X with an yearly interest rate of M after Y years is calculated based on a mathematical formula: Ret is S*(R+1)^P;
2. Tail recursive strategy: a variable can be used to accumulate the sum of investing money and its interest after each year;
3. Recursive and arithmetic_before strategy: the calculation of the profit of investing a sum of money goes back year after year to the first year of investment, then the profit of each year by summing up the invested money and its interest is determined;
4. Recursive and arithmetic_after strategy: first, the return is calculated recursively on a new period, then the new period is checked whether the old period is an increment of the new one. Following this strategy, a new period is not calculated, rather tested.

Table 1 shows the implementation of those four solution strategies in normal form. In the domain of creating UML diagrams, there exist many design guidelines and design patterns which have been defined by experts, e.g. [5, 9]. They can be used as solution strategies. An UML class diagram can not be assessed to be absolutely correct or wrong,

¹ Pure Prolog does not support high-order built-in predicates
but more or less useful according to a given use case. To assess the usefulness of a class diagram, design guidelines and design patterns are taken as the foundation. For example, the use case *Buy Item* is given: “This use case begins when a Customer arrives at a POST (point-of-sale terminal) checkout with items to purchase. The Cashier records the Universal Product Code (UPC) from each item. The POST determines the item price and adds the item information to the running sales transaction. The description and price of the current item are presented. On completion of item entry, the Cashier indicates to the POST that item entry is complete. The POST calculates and presents the sale total.” [9]

<table>
<thead>
<tr>
<th>Analytic</th>
<th>Tail recursive</th>
<th>Recursive &amp; arith_before</th>
<th>Recursive &amp; arith_after</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv(S,R,P,Ret):-</td>
<td>inv(S,_,P,S):-P=0.</td>
<td>inv(S,R,P,Ret):-P=0.</td>
<td>inv(S,__P,S):-P=0.</td>
</tr>
<tr>
<td>Ret is S*(R+1)^P.</td>
<td>inv(S,R,P,Ret):-P&gt;0,</td>
<td>NP is P-1,</td>
<td>inv(S,R,P,Ret):-P&gt;0,</td>
</tr>
<tr>
<td></td>
<td>NS is S*R+S,</td>
<td>Ret is NS + R*NS.</td>
<td>NS is S*R+S,</td>
</tr>
<tr>
<td></td>
<td>NP is P-1,</td>
<td></td>
<td>NP is P-1,</td>
</tr>
<tr>
<td></td>
<td>inv(NS,R,NP,Ret).</td>
<td></td>
<td>inv(S,R,NP,NS),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ret is NS+R*NS.</td>
</tr>
</tbody>
</table>

To design a diagram for the use case *Buy Item* one may have specified a *POSTNumber* attribute in the *Cashier* type (Figure 1, the worse diagram). According to the design guideline “*Not Attributes as Foreign Keys*”, this is undesirable because its purpose is to relate the *Cashier* to a *POST* object. The better way to express that a *Cashier* uses a *POST* is with an association, not with a foreign key attribute (see Figure 1, the better diagram). We can not say that the former design is wrong, but according to design guidelines it is less useful with respect to simplification and clarification.

Figure 1: Design decision: relate a type using an attribute or an association.

In addition to solution strategies which are based on design guidelines and design patterns, the design decision for a class diagram is mainly derived from the context of a given use case and individual justification [9, 10].

### 1.2 Constructs of Formal Languages

After choosing a solution strategy, the problem solver has to apply the constructs of the formal language of the domain being used. For example, to solve an arithmetic problem, the solver must be familiar with mathematic notations (+, -, *, :, =, <, >, etc.). There are multiple ways to apply these constructs to implement a solution. For example, to compare whether a variable X (X is an integer number) is greater than 1, we can express this comparison in many ways: e.g., \(X>1\), \(1<X\), \(X=2\), \(2=X\), \(X>0\), \(0<X-1\), \(1-X=0\), \(0>1-X\).

Similarly, in domain of programming a programmer has a choice of different constructs of a programming language. For the *Investment* problem stated above, different
semantic-preserving variants can be implemented for each solution strategy. For instance, the tail recursive strategy can be implemented by 512 variants in Prolog (Figure 2). In total, to solve the Investment problem, there are 1540 implementation possibilities.

In the domain of UML diagrams, a diagram also can be varied by applying different constructs resulting in diagrams which satisfy the same requirements. For instance, the diagram designer has the choice to model a noun phrase given in a problem statement using a class or an attribute of a class. Another example is when the designer has to decide to create a super class for more than two classes which share similar class attributes or class methods.

2. Classification of Problem Tasks

Based on solution strategies and a set of formal constructs of a learning domain, a solver is principally able to solve a problem in a potentially very large number of ways. However, many tutoring systems constrict the student's freedom [4] by over-specifying the problem statement or by prescribing a certain solution structure through the user interface. This results in a situation where the possibilities of applying different solution strategies and available formal constructs are narrowed down. Thus, problem tasks of a design domain can be simplified to a degree that it does no longer “represent the complexities of design tasks” [8] where aspects of ill-definedness play no significant role. In order to help ITS designers choose appropriate modeling approaches for ill-defined domains, we classify problem tasks into four classes with increasing size of solution space, which also can be viewed as a scale of ill-definedness.

2.1 Class 1: one solution strategy, one implementation

The problem tasks of this level can be solved only by one solution strategy (prescribed through the user interface) and have only one solution. For example, AnimalWatch, a system which helps students in solving arithmetic problems for 10 to 12-year-old students, poses the following problem [16, p.23]: A book says that one whale can eat 21 pounds of plankton in an hour. How many pounds can it eat in 7 hours? Enter your answer here: _______. Usually, solutions for such problem tasks are unique and thus, they are suited to recall basic knowledge of the domain being taught.
2.2 Class 2: one solution strategy, alternative implementation variants

On the second level, problem tasks can be solved by one solution strategy and the implementation of that strategy can vary. For example, the PAT system, which helps students learn introductory algebra, requests the student to solve problems by defining variables, rules and entering a correct algebraic expression and values of quantities into given rows of templates [16, p. 63]. The ANDES system, which supports students solving physics problems, also provides a similar user interface [16, p.156].

Another way to limit developing alternative solution strategies is to put more detailed information in the problem statement as the mentioned presentation about a tutoring system for database design shows [15]: “Sometimes students work in groups. Each group has a unique number and students have their student ids. A student who works in a group has a specific role within that group. The student may have different roles in various groups he/she belongs to.” The bold noun phrases are intended to be modeled as entities. The underlined verb phrase “work in” indicates an association between the two entities “student” and “group” and the underlined noun phrases such as number, student ids, and role can be identified to specify as attributes for entities or associations. Through this activity of mapping from elements of a problem description to constructs of database design (entities, relationships and associations), the student does not have to create a “design” at all because design decisions have been specified already in the problem statement.

A similar problem statement in the design domain yet requires no design activity: “Draw a UML class diagram for a school. A school is known by its name, address, and phone number and has one or more departments. Each department has a name and is assigned a number of instructors. Each instructor has a name and teaches several courses within the department. Each course is known by its name and course ID. A student has a name and student ID and attends a number of courses offered by the department. The school has a number of students can add students, remove students, add departments and remove departments…” [3]. The bold noun phrases e.g. “school”, “name”, etc. are intended to be selected and modeled as classes, attributes. The bold verb phrases such as “has one or more”, “add students”, “remove students” indicate class methods, associations, and association multiplicities to be specified. Such a problem description almost eliminates the design activity of the student, and leaving him to only map these bold phrases into constructs of UML. Although creating UML diagram is a problem which can be solved by many solution strategies and many diagrams can be created for the same requirements (as we showed in the previous section), such a problem description results in a small set of diagrams which are different instantiations of applying different UML constructs.

2.3 Class 3: alternative solution strategies

In this class of problem tasks the students are free to follow one of several alternative solution strategies and implement the solution according to their preferences. This kind of problem tasks is challenging to students because they have to make design decisions for developing a correct solution themselves, not guided by a predefined solution structure. In the case that a student’s solution does not satisfy the requirements of a given problem statement, appropriate feedback can only be given to the student if the system has a sensible reasonable hypothesis about the underlying solution strategy most likely employed by the student. Several systems providing problem tasks of this class have been developed, for examples: PROUST [7], a system supports programming in Pascal, Hong’s

---

2 In the original text of this problem task, the underlined phrases are light bold
programming tutor for Prolog [6]. However, these systems have not been evaluated in terms of their learning benefits yet.

2.4 Class 4: open-ended solution space and solutions are not verifiable

The task of developing a timetable information system is an example of a level four task. Although the goal of such an exercise seems to be quite obvious, a great number of design decisions have to be taken concerning the choice of appropriate data representations, a suitable method for heuristic search, the number and kind of input parameters considered, the presentation of the output, etc. Moreover, even the range of services the system could offer depends on individual preferences and is therefore open to debate.

Another example of this class is the problem of creating a class diagram for the use case *Buy Item* in the previous section. The solutions (class diagrams) for this problem cannot be formally verified as correct or incorrect, rather (at most) as useful or not. Due to individual preferences which are normally not specified in a design problem (or a use case), the space of solutions for such problem tends to be open-ended.

3. Applicability of Modeling Approaches to Classes of Problem Tasks

Currently, model-tracing and constraint-based modeling are the two most prominent approaches which have been applied successfully in building tutoring systems for different domains. In a model-tracing system, error diagnosis and instruction are carried out on the basis of an expert model. This model represents one or more *ideal* solution paths to a given problem. A solution path consists of many pairs of situation and action. Whenever a student solution deviates from the expert model, the system identifies the situation where the student has carried out a wrong action. Since modeling a solution path is laborious, normally just one or two solution paths are integrated in the expert model [16, p.85]. Modeling all possible solution paths is a substantial undertaking. Thus, this approach is appropriate for problem tasks of classes 1 and 2 which can be solved by one solution strategy.

While model-tracing systems are based on ideal solution models, CBM systems are built from sets of constraints. A constraint represents a domain principle or specifies a task requirement [14]. A CBM system does not require modeling all alternative solution paths for a problem task. Instead, an *ideal* solution is specified for each problem task, and constraints are used to model the principles of the application domain, and to compare the student solution with the semantic elements of the ideal solution. In order to be able to deal with alternative ways of applying formal constructs in order to satisfy a situation, a constraint should include all possible valid solutions, instead of modeling different particular sequences of problem solving steps. For example, in the domain of defining database queries, SQL has three completely different ways for retrieving data from multiple tables by specifying required tables : 1) in the FROM clause, 2) in a JOIN clause, and 3) in a nested SELECT. A constraint might be formulated considering the existence of required tables in different cases as follows [13, p.35]:

IF in the ideal solution, Table A exists in the FROM clause
THEN in the student solution, Table A exists in the FROM clause,
    OR Table A exists in the WHERE clause,
    OR Table A exists in a JOIN clause.
A solution violates constraints if either it does not adhere to principles of the domain or does not satisfy the requirements of a given problem task. Although the CBM approach is able to identify different implementations of formal constructs to achieve a satisfied situation, it is inherently not in a position to identify the intention underlying a solution for problem tasks because constraints can be specified to cover implementation variability. Furthermore, feedback returned from a CBM system might mislead students as studies claimed [13, 8] because a constraint does not encapsulate information about a solution strategy and constraint violation is based on an ideal solution. Therefore, this approach is applicable to develop tutoring systems which provide problem tasks of classes from 1 to 2.

The weighted constraint-based approach (W-CBM) has been proposed in order to close this fundamental lack of CBM. W-CBM has been realized in building a tutoring system for logic programming [11]. The authors enriched the traditional constraint approach with constraint weights which indicate the importance of each constraint. This kind of information serves two essential purposes: 1) identifying the solution strategy underlying a students’ solution, and 2) prioritizing feedback according to the importance of each diagnostic information. Instead of expert models (as used in a model-tracing system), a W-CBM system uses a so-called semantic table which specifies semantic elements required by alternative solutions strategies. Information from this table is used to check the semantic correctness of students’ solutions, regardless of solution strategies. In order to identify the solution strategy underlying a solution, the system iterates through each possible solution strategy and hypothesizes that the student has implemented that strategy. Then, the system invokes relevant constraints to assess the semantic correctness of the solution by considering semantic elements specified for that strategy. The hypothesized solution strategy with the least constraint violations in terms of constraint weights is considered as the most plausibly implemented by the student. Since such a W-CBM approach is able to identify intention underlying a student solution, it is applicable to building tutoring systems which provide problem tasks of the class 3.

The W-CBM approach also has been attempted in the domain of designing UML class diagrams [10]. However, this system has not been completed and reported on its learning benefits. In this system, the semantic table contains of mandatory and optional elements, which represent design guidelines/design patterns and individual justifications, respectively. Constraints considering mandatory elements are regarded more severe than constraints which encapsulate optional elements. If a class diagram submitted by a student violates constraints, it indicates that he/she has not adhered design guidelines/patterns or not followed individual advisories. Hence, it is promising that the W-CBM is able to be applied to developing ITS for problem tasks of the class 4.

4. Conclusion

We have identified two variability levels of solutions for a problem task: solution strategy and constructs of a formal language. Based on these two levels, this paper has classified problem tasks into four classes with respect to solution space: 1) problem tasks which have only one solution, 2) problem tasks for which only one solution strategy is available and the strategy can be implemented by different variants, 3) problem tasks which can be solved by alternative solution strategies and each can be implemented by different ways, and 4) problem tasks whose solution space is open-ended and whose solution cannot be verified as correct or not.

Based on this classification, we have identified appropriate modeling approaches for each class of problem tasks. The model-tracing and constraint-based approaches are suitable for the classes 1 and 2, where the weighted constraint-based approach is useful for the class 3. Since the latter approach is capable with constraint weights which can be used
to represent a mandatory design guideline (severe elements) or an individual preference (weak elements), we predict that it is also applicable to class 4 of problem tasks.

References