An extensible domain-specific language for describing problem-solving procedures

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2017-007
April 2017

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

An extensible domain-specific language for describing problem-solving procedures

Bastiaan Heeren[1] and Johan Jeuring[1,2]

[1] Faculty of Management, Science & Technology, Open University of the Netherlands
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
[2] Department of Information and Computing Sciences, Universiteit Utrecht
`Bastiaan.Heeren@ou.nl` `J.T.Jeuring@uu.nl`

Abstract. An intelligent tutoring system (ITS) is often described as having an inner loop for supporting solving tasks step by step, and an outer loop for selecting tasks. Many task domains have problem-solving procedures that express how tasks can be solved by applying steps or rules in a controlled way. In this paper we collect established ITS design principles, and use the principles to compare and evaluate existing ITS paradigms with respect to the way problem-solving procedures are specified. We argue that problem-solving procedures need an explicit representation, which is missing in most ITSs. We present an extensible domain-specific language (DSL) that provides a rich vocabulary for accurately describing procedures. We give three examples of tutors from different task domains that illustrate our DSL approach and highlight important qualities such as modularity, extensibility, and reusability.

1 Introduction

Intelligent tutoring systems (ITSs) are large and complex software systems that typically take many years to build and improve. Full-blown ITSs offer a range of functionality, including components for providing hints and feedback, modeling student skills, tutoring strategies, course administration, tools for authoring content, etc. Developing an ITS and its instructional content requires many areas of expertise, such as skills in educational design, software engineering, user experience, AI techniques, cognitive psychology, the task domain, and more. Software architecture and solid design principles help us to deal with this complexity.

In this paper we look at the step-based inner loop of an ITS [25] that is responsible for giving feedback and providing hints. An ITS with an inner loop, such as Andes [27] or PAT [13], lets a user enter steps that she would take when solving problems normally, without using a digital tutor. Such systems are almost as effective as human tutors [26]. For many task domains, a problem-solving procedure can be specified to provide hints and feedback for the inner loop. An example of such a procedure is adding two fractions [17], which consists of the following steps: (1) find the lowest common denominator (LCD), (2) convert fractions to LCD as denominator, (3) add the resulting fractions, and (4) simplify the final result. A more detailed analysis of these steps reveals that adding two fractions that already have the same denominator is a special case (the first two steps can be skipped), other multiples of the denominators also work (although this is not preferred), step 2 consists of two smaller conversions, and the simplification (step 4) is not always needed. How is this procedure encoded in software? Can we access the high-level structure of the procedure, e.g. for presenting an outline of the problem-solving procedure? This paper addresses these questions by analyzing the relevant literature, and proposing an alternative.

The services that are offered by different ITSs to support the inner loop are very similar [25], but their internal structures and representations (i.e., *how* they provide these services) are not. Non-functional requirements, also called quality attributes, are concerned with how a system provides its functionality [8], and they become increasingly important for larger systems. ITSs are no exception: the internal structures and models for representing knowledge determine important non-functional qualities such as reusability, modularity, and maintainability.

Various approaches and paradigms for intelligent tutoring exist. In particular, there are model-tracing tutors [2], example-tracing tutors [1], constraint-based tutors [16], and data-driven tutors [14]. A significant difference between the paradigms lies in the way the expert knowledge necessary for following the steps of a student, is specified. Authoring tools built on top of these paradigms simplify the construction of an ITS: they hide the underlying software layer from the developer of an ITS, sometimes completely removing the need for programming skills [19]. Using such a tool, the developer focuses on the tutoring interface and the expert knowledge necessary for the tutor. Murray [19] clearly describes the design space for authoring tools, and identifies design trade-offs. For example, the advantage of an easy-to-use authoring environment often comes together with reduced expressiveness, since the programming layer is hidden for the developer.

According to Nkambou et al. [20] there is a large biodiversity or even a Tower of Babel in the field of authoring ITSs. As a result, after thirty years, existing solutions are still not widely shared in the field, making it difficult to find adequate building blocks and guidance to build an ITS. The first contribution of this paper is a critical evaluation of how problem-solving procedures are specified in various ITS paradigms and authoring tools, based on reported design principles. More specifically, we argue that it is important to have an explicit knowledge representation for problem-solving procedures, which is absent in most ITSs. We present an extensible domain-specific language (DSL) that provides a rich vocabulary for accurately describing procedures and demonstrate how this has been used to develop several tutoring systems, which is our second contribution.

This paper is organized as follows. Based on the scientific literature on tutoring systems and on software quality, Section 2 introduces a number of design principles and best-practices for developing ITSs and authoring tools. Section 3 describes the most common ITS paradigms and evaluates how they follow the principles. In Section 4 we propose to define explicit knowledge models for representing problem-solving procedures in an extensible DSL. We present three examples of task domains (Section 5) for which explicit problem-solving procedures have been constructed successfully. Finally, Section 6 concludes this paper.

Anderson et al. (cognitive tutors) [3]	
– Represent student competence as a production set	(a)
– Communicate the goal structure underlying the problem solving	(b)
– Promote an abstract understanding of the problem-solving knowledge	(c)
– Adjust the grain size of instruction with learning	(d)
Beeson (algebra and calculus tutor) [4]	
– Cognitive fidelity	(e)
– Glass box computation	(f)
– Customize step size to individual user	(g)
Murray (Eon authoring tools) [18]	
– Represent instructional content and instructional strategies separately	(h)
– Modularize the instructional content for multiple use and reuse	(i)
– Explicitly represent abstract pedagogical entities (such as topics)	(j)
Murray (analysis of authoring tools) [19]	
– Instructional content should be modular and reusable	(k)
– Authoring tools should provide customization, extensibility, and scriptability	(l)
– Include customizable representational formalisms	(m)
Aleven et al. (example-tracing tutors) [1]	
– Support authoring of effective, intelligent computer-based tutors	(n)
– Facilitate the development of tutors across a range of applications domains	(o)
– Support cost-effective tutor development	(p)
– Create tutors that are easy to maintain	(q)

Table 1. Design principles for the inner loop

2 Design principles for the inner loop

The inner loop of an ITS is about the steps within a task and the services that are available for tutoring [25]. As VanLehn points out, the behavior of systems with an inner loop is surprisingly similar, but their internal structures can be very different. We first collect design principles from successful ITSs and authoring tools that provide insight into how to construct software that supports an inner loop. We have found five papers [3, 4, 18, 19, 1] that explicitly describe such design principles. We only include principles that are particularly relevant for the inner loop. Quite a few more papers discuss authoring tools for ITSs (e.g. [17, 7, 20]), but the former set specifically gives design principles for specifying the inner loop. The set of selected design principles is listed in Table 1.

The internal structure of an ITS is often described in terms of four components [21]: the expert knowledge, a student model, tutoring or instructional strategies, and the user interface. This decomposition into four parts is a separation of concerns. In particular, instructional content and instructional strategies are separated (h). The principles *glass box computation* (f: the student can see how the system solves the problem step by step) and *cognitive fidelity* (e: the system solves the problem in the same way as the student) are clear guidelines for how to support the inner loop.

An accurate model of the target skill is needed [3], and this model should be *abstract* (a) and *explicit* (j). Preferably, such a model is also *modular* and *reusable* (i, k). Modular content can be used for multiple instructional purposes [19], and thus promotes reuse. Repetitive and template-like content should be avoided: an *expressive* representation helps to concisely describe content. Having modular procedures with an explicit representation simplifies decomposing a problem into a set of goals and subgoals (b), and may help to promote an abstract understanding of the procedure (c).

The *generality* of a tool or technique is one dimension in the design space [19] that determines how many application domains can be supported (o). However, a tool cannot anticipate everything an author will want, and this is even more problematic for general tools. Authoring tools should therefore be *customizable*, *extensible*, and *scriptable* (l, m), just as all modern design and authoring software. Important for the inner loop is the ability to customize the step size (from many small rules to a few powerful rules) and the grain size of instruction (d, g). Such customizations can be steered by authors, students, the student model, or a combination.

The ultimate goal is to develop ITSs that support effective tutoring (n), which requires a *flexible* inner loop that can deal with multiple solution paths [1, 27]. Aleven et al. [1] explain that flexibility and avoiding repetitious authoring tasks make ITS maintenance easier (q). A final consideration is that ITS development should be cost-effective (p), especially because estimations of 200-300 development hours per hour of instruction are not uncommon [19]. Proven tactics to reduce the development time are authoring tools that simplify content creation, and reuse (e.g. by better interoperability between systems). Reusability is the key to improving productivity and quality [8].

3 ITS paradigms

In this section we present approaches for developing an ITS and we discuss how problem-solving procedures can be defined or authored. All of these approaches have been successfully used to develop systems that have been tested and used in practice. We evaluate the approaches using the design principles from the previous section, and highlight their limitations.

Cognitive tutors (based on production rules). Many cognitive tutors have been developed that are based on the ACT-R theory for simulating and understanding human cognition [2, 3]. In this theory, declarative knowledge (facts) and procedural knowledge (problem-solving behavior) are distinguished. An ACT-R tutor uses an ideal student model to trace the steps of a student. This process is called model tracing. The ideal student model is defined as a set of production rules in the form of if-then statements. There are no further facilities for structuring the production rules. The ACT-R software framework is developed in the Lisp programming language, which can be used to introduce more structure.

The set of production rules describing the ideal student model in an ACT-R tutor contains an implicit description of the problem-solving procedure. It is in general very hard to extract an *explicit* description of the procedure from this model. When a large number of rules are involved, understanding the interactions between multiple rules affected by the same facts can become very difficult [24]. Subprocedures that are encoded as sets of production rules cannot be combined without carefully considering the possible interactions. We conclude that (sets of) production rules are not *modular* and not straightforward to *reuse*.

Model-tracing tutors (based on procedures). The Extensible Problem Specific Tutor (xPST) system [7] is an authoring environment that enables non-programmers to create an ITS on top of an existing software application by providing instruction inside this application. 'Extensible' in the system's name refers to the plug-in architecture for connecting to different software applications. An ITS is specified in an xPST instruction file, which contains a sequence section (among other sections) for specifying the problem-solving procedure. Four types of operators are available for the developer: THEN, OR, AND, and UNTIL. Similarly, ASTUS [22] represents hierarchical procedure knowledge as a graph.

In both systems, procedures are specified *explicitly* and they are *modular*. Unfortunately, the set of operators is not *expressive* enough for describing procedures in more complex domains without repetition, and there is no way to easily *extend* this set with more operators or traversals.

Constraint-based tutors. The constraint-based tutoring paradigm [16, 17, 23] simplifies ITS development by focusing on conditions that should hold for correct solutions, rather than defining how to reach such a solution. In this paradigm, constraints have three parts: a relevance condition (when is it applicable), a satisfaction condition (what should hold), and a feedback message that is reported if the constraint is violated. The constraint-based approach can be very effective, especially for domains in which there is no clear path to reach a correct solution. Although a constraint-based tutor can evaluate solutions, it is not capable of actually solving the given problem itself. Hence, the *cognitive fidelity* (e) and *glass box computation* (f) design principles are violated by this paradigm.

Example-tracing tutors. The Cognitive Tutor Authoring Tools (CTAT) [1] can be used to create cognitive tutors without programming. The main idea behind these tutors is that worked-out examples are used for tracing student steps. Example-tracing tutors target particular tasks (e.g. solve $3x-6=8+x$) instead of a class of similar tasks (e.g. linear equations) and thus prefer usability and a low entry level over productivity for trained users. The worked-out examples are recorded in a behavior graph that contains sequences of steps. These graphs can be generalized to increase the *flexibility* and recognize more solution paths, for instance by making steps optional or unordered.

Behavior graphs make the problem-solving steps *explicit*, but there is no general problem-solving procedure. The facilities for generalizing behavior graphs are key in making the approach viable [1], but are provided on an ad-hoc basis and are limited in *expressiveness*. The generalization step complicates creating a tutor (for CTAT's intended users), which once again demonstrates the inherent design trade-off between flexibility and usability [19].

Data-driven tutors. An approach that is recently gaining more and more attention is to use historical student data for developing an ITS [14]. Successful solutions from the past can be used to provide feedback and hints for students in the present, which circumvents the need to create an expert model. A data-driven tutoring system can be bootstrapped by experts providing missing data. The data-driven approach has proven to work well in combination with AI and machine-learning techniques for learning an expert model by demonstration.

Data-driven ITSs have no *explicit* expert model, which makes it hard for instructors to *customize* a tutor. Instructors cannot express preferences, such as shorter solution paths that are not found by the average student.

4 Problem-solving procedures

We now present a different approach to developing tutoring systems, which is based on having an *explicit* representation (model) for problem-solving procedures. This representation is based on operators that allows simple procedures to be combined into more complex, composite procedures (in the spirit of the composite design pattern). For instance, sequence ('first do A, then B', denoted $A ; B$) and choice ('do A or B', denoted $A \mid B$) are operators that can be used to create composite procedures. The primitive procedures (i.e., the leaf nodes in the tree structure) are the steps or production rules that may or may not apply in a particular situation. A fixed point construct is used for expressing recursive procedures. Because of the operators, the models for describing problem-solving procedures are *modular*, and therefore also *reusable* instructional content.

An advantage of having an explicit model is that it can be used for multiple purposes and interpreted in different ways: the model can be executed step by step, used to generate a student model, visualized (e.g. to increase the understanding), sent to another tool, etc. For an ITS, the stepwise execution of a procedure is a particularly important interpretation of the model since this is needed for generating next-step hints and worked-out solutions, and for tracing student steps (services of the inner loop [25]). Figure 1 presents trace-based semantics $\mathcal{T}$ for core procedures. Each trace represents a sequence of steps, where symbol $\checkmark$ denotes successful termination of the procedure. From these traces, alternative next steps can be calculated, and worked-out solutions can be constructed. The technical details can be found elsewhere [10, 9].

We argue that this approach is *extensible*: new composition operators can be added easily by defining their stepwise execution $\mathcal{T}$, or by expressing the operator in terms of existing operators. For example, performing a procedure s zero or more times (the Kleene star) can be defined by $\mathit{many}\ s = \mu x.(s ; x) \mid \mathit{succeed}$. We have defined many more useful operators (see Table 2), such as for interleaving procedures, for making some part optional, and for various kinds of generic traversals (for domains that have a notion of subterms). Such extensibility is essential for supporting a diversity of task domains.

$$\begin{array}{lll}
\mathcal{T}(s\,;\,t) & = \{x \mid x \in \mathcal{T}(s), \checkmark \notin x\} \cup \{xy \mid x\checkmark \in \mathcal{T}(s), y \in \mathcal{T}(t)\} & \text{(sequence)}\\
\mathcal{T}(s \mid t) & = \mathcal{T}(s) \cup \mathcal{T}(t) & \text{(choice)}\\
\mathcal{T}(\mu x.f(x)) & = \mathcal{T}(f\,(\mu x.f(x))) & \text{(fixed point)}\\
\mathcal{T}(r) & = \{\epsilon, r, r\checkmark\} & \text{(rule)}\\
\mathcal{T}(\mathit{succeed}) & = \{\epsilon, \checkmark\} & \text{(success)}\\
\mathcal{T}(\mathit{fail}) & = \{\epsilon\} & \text{(failure)}
\end{array}$$

Fig. 1. Trace-based semantics for problem-solving procedures

operator	description	operator	description
$s\,;\,t$	first s, then t	*not s*	succeeds if procedure s is not applicable
$s \mid t$	either s or t		
succeed	ever succeeding procedure	*repeat s*	apply s as long as possible
fail	ever failing procedure	*repeat1 s*	as *repeat*, but at least once
$\mu x.f(x)$	fixed point combinator	*try s*	apply s once if possible
label ℓ *s*	attach label ℓ to s	$s \triangleright t$	apply s, or else t
many s	apply s zero or more times	*somewhere s*	apply s at some location
many1 s	apply s one or more times	*bottomup s*	search location bottom-up
option s	either apply s or not	*topdown s*	search location top-down

Table 2. Selection of composition operators

The composition operators can be considered a simple domain-specific language (DSL) [5] for expressing problem-solving procedures. The DSL captures common patterns that are found in procedures and provides a vocabulary for these patterns. The rich vocabulary combined with the possibility to add more operators make the language *expressive*. The DSL helps authors to articulate problem-solving processes in their task domain.

For example, consider the procedure for adding two fractions from the introduction, which uses four rules and can be expressed in the DSL as:

`FindLCD` ; *many* (*somewhere* `Convert`) ; `Add` ; *try* `Simplify`

This procedure produces the following stepwise solution for $\frac{1}{2} + \frac{4}{5}$:

$$\tfrac{1}{2} + \tfrac{4}{5} \overset{\texttt{FindLCD}}{\Longrightarrow} \tfrac{1}{2} + \tfrac{4}{5} \overset{\texttt{Convert}}{\Longrightarrow} \tfrac{5}{10} + \tfrac{4}{5} \overset{\texttt{Convert}}{\Longrightarrow} \tfrac{5}{10} + \tfrac{8}{10} \overset{\texttt{Add}}{\Longrightarrow} \tfrac{13}{10} \overset{\texttt{Simplify}}{\Longrightarrow} 1\tfrac{3}{10}$$

The step for finding the LCD calculates the value used by `Convert`, and this step may or may not show up in a learning environment, depending on step size and the exact user interface. Similarly, sub-procedure *many* (*somewhere* `Convert`) can be collapsed into a single step, which shows how the step size can be adjusted.

Our DSL for problem-solving procedures is very similar to other formalisms for describing sequences. The DSL was mainly inspired by context-free grammars and Hoare's communicating sequential processes (CSP) [11]. Because of the similarity with these formalisms, we can reuse techniques and definitions from these formalisms, such as parsing and interleaving, and apply these in the context of an ITS. Similar languages have been used in different application domains for describing workflows, term rewriting, and proof tactics.

5 Examples of problem-solving procedures in ITSs

We illustrate how we have used the DSL for problem-solving procedures in a number of tutoring systems that have been used in a classroom setting. These systems cover three completely different domains: math, introductory programming, and practicing communication skills. For each system, we describe how the procedures are developed, and we discuss what the advantages are of having explicit procedures. Where applicable, we discuss non-functional qualities of the system such as reusability, interoperability, and customizability.

5.1 Math tutor

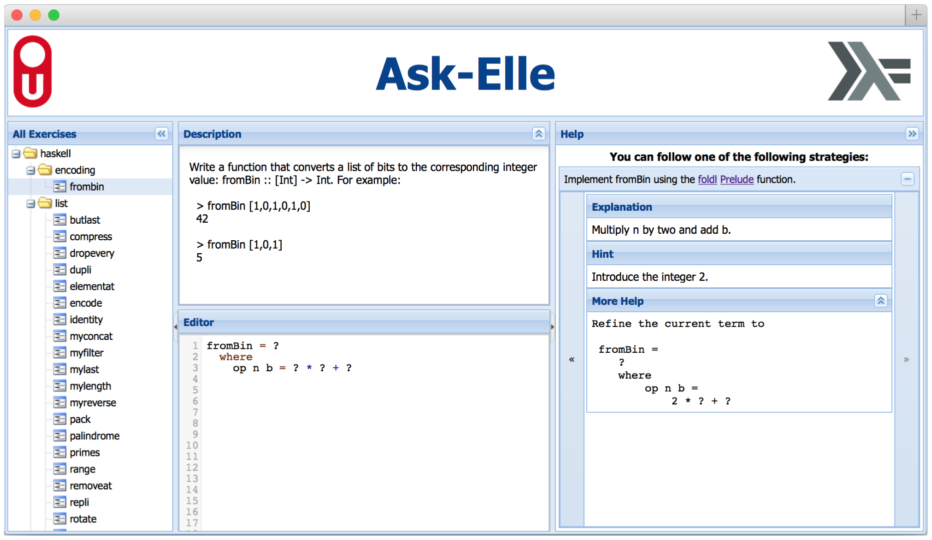
We have constructed an expert knowledge module for (high school) mathematics that covers many topics, such as solving equations, calculations, and linear algebra. The problem-solving procedures are defined in an embedded domain-specific language that offers the operators in Table 2 (and many more), and also provides access to the underlying programming language. The math domain allows for a lot of reuse: for example, the procedure for solving a linear equation is part of the procedure for quadratic equations. Many tasks have alternative methods for solving, resulting in different configurations that can be used. Such variation is easy to deal with in the DSL.

The expert knowledge module for math is used by at least three external learning environments, which provides some evidence for the interoperability of our approach. The module provides a number of request-response feedback services [9] that are derived from the problem-solving procedures. These services are used by the learning environments in different ways and with different choices in how and when feedback and hints are offered. For example, one environment uses the hierarchical structure of a problem-solving procedure to automatically decompose a problem into a group of subproblems when a student is not able to solve the complete task. The Digital Mathematics Environment[1] has an authoring tool that allows content developers to tailor the feedback and hints that are calculated by our expert knowledge module. For example, feedback messages can be customized and certain inner loop services can be enabled or disabled.

5.2 Functional programming tutor

Programs are written step by step: starting with an empty (or skeleton) program, the program gradually becomes more defined. The Ask-Elle programming tutor [6] for Haskell was developed to help students with writing typical beginner's programs by giving feedback and hints (see Figure 2). Students can use holes (?) in their functions for parts that are not yet defined. The steps towards a full solution are refinement rules that replace a hole by some expression that may contain new holes.

[1] `https://www.dwo.nl/site/index_en.html`



Write a function that converts a list of bits to the corresponding integer value: fromBin :: [Int] -> Int. For example:

```
> fromBin [1,0,1,0,1,0]
42

> fromBin [1,0,1]
5
```

```haskell
fromBin = ?
  where
    op n b = ? * ? + ?
```

You can follow one of the following strategies:

Implement fromBin using the foldl Prelude function.

Multiply n by two and add b.

Introduce the integer 2.

```
Refine the current term to

 fromBin =
    ?
    where
        op n b =
            2 * ? + ?
```

Fig. 2. The Ask-Elle programming tutor for learning Haskell

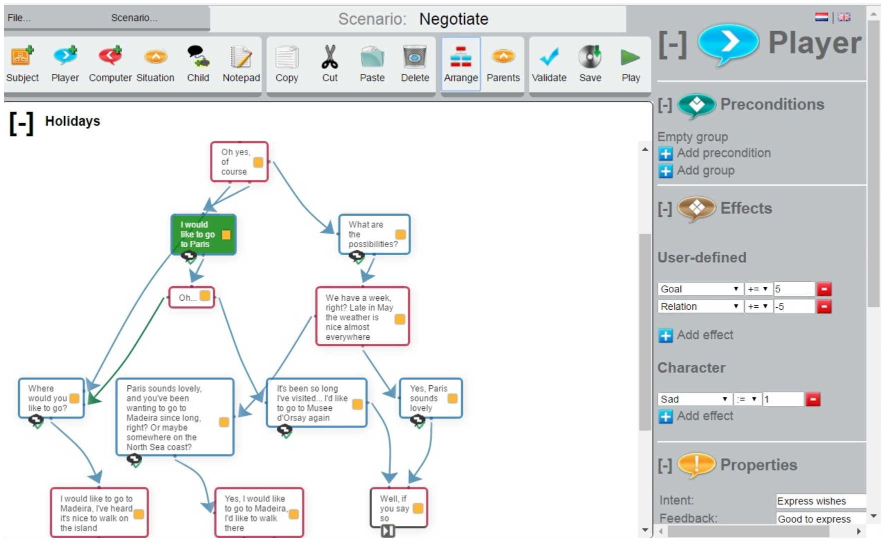
Teachers can specify new programming tasks by providing one or more model solutions for the task: these solutions follow good programming practice and are written in the target language. Writing model solutions does not require any knowledge about the structure or inner workings of the tutoring system. Model solutions can be annotated by the teacher to customize and further specialize the tutoring, for instance, to attach specific feedback messages to parts in the solution, or to indicate that a certain language construct must be used.

For each model solution, a problem-solving procedure is generated in the DSL, and these procedures are combined as choices. The final procedure can be used to provide hints or to trace student steps, automatically disambiguating between the different model solutions. Tracing student programs in a programming tutor is difficult because there are many ways and variations to define something. This variation is partly dealt with during generation: alternative solutions are generated for standard functions and for some language constructs. The remaining variation is tackled by aggressively normalizing the student program.

When a student takes off-path steps, the tutor can no longer guarantee that these steps can lead to a correct solution. In such cases, the tutor uses testing to decide about the correctness of the solution. This illustrates that a constraint-based (testing) approach can complement the problem-solving procedures.

5.3 Serious game for communication skills

The serious game Communicate! [12] was developed for practicing interpersonal communication skills between a healthcare professional and a patient. The player is offered a list of alternative sentences during a consultation with a virtual



Scenario: Negotiate

[-] Holidays

Oh yes, of course

I would like to go to Paris

What are the possibilities?

Oh...

We have a week, right? Late in May the weather is nice almost everywhere

Where would you like to go?

Paris sounds lovely, and you've been wanting to go to Madeira since long, right? Or maybe somewhere on the North Sea coast?

It's been so long I've visited... I'd like to go to Musee d'Orsay again

Yes, Paris sounds lovely

I would like to go to Madeira, I've heard it's nice to walk on the island

Yes, I would like to go to Madeira, I'd like to walk there

Well, if you say so

Empty group

User-defined

Goal += 5
Relation += -5

Character

Sad := 1

Intent: Express wishes
Feedback: Good to express

Fig. 3. The Communicate! authoring tool for developing communication scenarios

patient. These sentences are the steps in the inner loop. During the consultation, the player receives feedback by means of emotions shown by the virtual patient and the patient's reaction. After the consultation is finished, the player gets a final score and feedback on how appropriate each step was.

Communicate! has a specialized scenario editor (see Figure 3), which enables non-technical communication trainers to develop and test their communication scenarios. This authoring tool allows trainers to develop a graph-like structure for a particular scenario, but the tool also offers scenario authors some domain-specific features that are typical for consultations, such as:

- conditions under which certain options are offered or not;
- parts of consultations that may be interleaved in any order;
- or that (part of) a consultation may be stopped at any point.

These scenarios are translated to stepwise procedures that control the sequencing. These procedures in the DSL give us very fine control over the structure of consultation compared to dialog trees and avoid repeating subtrees.

6 Conclusions and future work

We have proposed to use a domain-specific language to describe problem-solving procedures for an ITS. This language is extensible, and enables content authors to articulate procedures in a modular and reusable way. We have exemplified the DSL approach and its feasibility by describing three tutors for different task domains that are based on this language. Other domains for which the DSL was used are proposition logic, logical equivalence, axiomatic proofs [15], evaluating expressions, and microcontroller programming. The DSL provides an explicit representation for problem-solving procedures, which is missing in other ITS paradigms. The need for an explicit, modular, and expressive representation is supported by several design principles that are reported in the literature.

The DSL is general and independent of the task domain: it works best for domains in which the order of steps must be controlled, with various degrees of freedom in how strict the order must be. The procedures capture deep domain knowledge that makes further reasoning steps possible. With respect to Murray's design space [19], our approach is positioned more towards productivity and expressiveness than learnability. However, the specialized scenario editor for the communication skills serious game, which is targeted at scenario authors without technical skills, shows that the DSL can also be used as an intermediate layer between a graphical editor and a tutoring system.

In the future, we want to simplify the authoring of procedures and provide more guidance to authors. We want to approach this problem from several angles (e.g. with graphical editors) because there is no 'one size fits all' solution. Ideally we can use ITS techniques to generate feedback for content authors. We are also interested in interpreting problem-solving procedures in new ways, for example, to present procedures visually, to transform procedures into simpler or more efficient procedures (using algebraic laws for the composition operators), and to calculate the coverage of procedures given a set of stepwise solutions.

We conclude the paper with observing that there is a trend away from having problem-solving procedures in an ITS. The procedures are either absent (in constraint-based and data-driven tutors), or very restricted (in authoring tools): the motivation is mostly to make ITS development more cost-effective. We claim that investing in techniques that improve interoperability between systems and large-scale reuse is a good alternative strategy that deserves more attention. The DSL we presented is a small step in that direction.

References

1. V. Aleven, B.M. McLaren, J. Sewall, and K.R. Koedinger. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *Journal of AIED*, 19(2):105–154, 2009.
2. J.R. Anderson, C.F. Boyle, A.T. Corbett, and M.W. Lewis. Cognitive modeling and intelligent tutoring. *Artificial intelligence*, 42(1):7–49, 1990.
3. J.R. Anderson, A.T. Corbett, K.R. Koedinger, and R. Pelletier. Cognitive tutors: lessons learned. *Journal of the Learning Sciences*, 4(2):167–207, 1995.
4. M.J. Beeson. Design principles of MathPert: Software to support education in algebra and calculus. In N. Kajler, editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer, 1998.
5. A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.
6. A. Gerdes, B. Heeren, J. Jeuring, and L.T. van Binsbergen. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *Journal of AIED*, pages 1–36, 2016.
7. S.B. Gilbert, S.B. Blessing, and E. Guo. Authoring effective embedded tutors: An overview of the extensible problem specific tutor (xPST) system. *Journal of AIED*, 25(3):428–454, 2015.

8. I. Gorton. *Essential software architecture*. Springer, 2006.
9. B. Heeren and J. Jeuring. Feedback services for stepwise exercises. *Science of Computer Programming*, 88:110–129, 2014.
10. B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.
11. C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.
12. J. Jeuring, F. Grosfeld, B. Heeren, M. Hulsbergen, R. IJntema, V. Jonker, N. Mastenbroek, M. van der Smagt, F. Wijmans, M. Wolters, and H. van Zeijts. Communicate! — a serious game for communication skills. In *EC-TEL 2015*, volume 9307 of *LNCS*, pages 513–517. Springer, 2015.
13. K.R. Koedinger, J.R. Anderson, W.H. Hadley, and M.A. Mark. Intelligent tutoring goes to school in the big city. *Journal of AIED*, 8:30–43, 1997.
14. K.R. Koedinger, E. Brunskill, R.S.J.d. Baker, E.A. McLaughlin, and J. Stamper. New potentials for data-driven intelligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.
15. J. Lodder, B. Heeren, and J. Jeuring. Generating hints and feedback for Hilbert-style axiomatic proofs. In *SIGCSE 2017*, pages 387–392, 2017.
16. A. Mitrovic, B. Martin, and P. Suraweera. Intelligent tutors for all: The constraint-based approach. *Intelligent Systems, IEEE*, 22(4):38–45, 2007.
17. 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. *Journal of AIED*, 19(2):155–188, April 2009.
18. T. Murray. Authoring knowledge-based tutors: Tools for content, instructional strategy, student model, and interface design. *Journal of the Learning Sciences*, 7(1):5–64, 1998.
19. T. Murray. An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art. In T. Murray, S.B. Blessing, and S. Ainsworth, editors, *Authoring Tools for Advanced Technology Learning Environments*, pages 491–544. Springer, 2003.
20. R. Nkambou, J. Bourdeau, and V. Psyché. Building intelligent tutoring systems: An overview. In R. Nkambou et al., editor, *Advances in Intelligent Tutoring Systems, SCI 308*, pages 361–375. Springer, 2010.
21. H.S. Nwana. Intelligent tutoring systems: an overview. *AI Review*, 4(4):251–277, 1990.
22. L. Paquette, J.-F. Lebeau, G. Beaulieu, and A. Mayers. Designing a knowledge representation approach for the generation of pedagogical interventions by MTTs. *Journal of AIED*, 25(1):118–156, 2015.
23. R. Sottilare, A. Graesser, X. Hu, and K. Brawner, editors. *Design Recommendations for Intelligent Tutoring Systems. Volume 3: Authoring Tools and Expert Modeling Techniques*. Adaptive Tutoring Series. 2015.
24. R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.
25. K. VanLehn. The behavior of tutoring systems. *Journal of AIED*, 16(3):227–265, 2006.
26. K. VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011.
27. K. VanLehn, C. Lynch, K. Schulze, J.A. Shapiro, R. Shelby, L. Taylor, D. Treacy, A. Weinstein, and M. Wintersgill. The Andes physics tutoring system: Lessons learned. *Journal of AIED*, 15(3):147–204, 2005.