2015

Authoring Effective Embedded Tutors: An Overview of the Extensible Problem Specific Tutor (xPST) System

Stephen B. Gilbert
Iowa State University, gilbert@iastate.edu

Stephen B. Blessing
University of Tampa

Enruo Guo
Iowa State University

International Journal of Artificial Intelligence in Education Volume# (YYYY) Number
IOS Press

Authoring Effective Embedded Tutors: An Overview of the Extensible Problem Specific Tutor (xPST) System

Stephen B. Gilbert, *Industrial & Manufacturing Systems Engineering, Iowa State University, Ames, IA 50011, USA*
gilbert@iastate.edu

Stephen B. Blessing, *Psychology, University of Tampa, Tampa, FL 33606 USA*
sblessing@ut.edu

Enruo Guo, *Computer Science, Iowa State University, Ames, IA 50011, USA*
enruoguo@iastate.edu

Abstract. The Extensible Problem Specific Tutor (xPST) allows authors who are not cognitive scientists and not programmers to quickly create an intelligent tutoring system that provides instruction akin to a model-tracing tutor. Furthermore, this instruction is overlaid on existing software, so that the learner's interface does not have to be made from scratch. The xPST architecture allows for extending its capabilities by the addition of plug-ins that communicate with additional third-party software. After reviewing this general architecture, we describe three major implementations that we have created using the xPST system, each using different third-party software as the learner's interface. We have conducted three evaluations of authors using xPST to create tutoring content, and these are considered in turn. These evaluations show that xPST authors can quickly learn the system, and can efficiently produce successful embedded instruction.

Keywords. Authoring Tool, Cognitive Tutor, Model-Tracing Tutor, xPST

INTRODUCTION

We have developed the Extensible Problem Specific Tutor (xPST) authoring tool to address the need of how to quickly create computer-based instruction that follows the behavior of model-tracing tutors. It does this by two main methods: 1) reducing if not eliminating the need for programming knowledge in constructing such a tutor, and 2) providing the capability to use third-party software as the student interface. Both of these factors together allow a person such as an instructional designer the ability to create computer-based instruction for their students that ensure their students stay on path while solving the problems, while providing appropriate and customized hints and just-in-time error messages along the way, just like a model-tracing tutor.

Even though model-tracing tutors have been shown to be effective in labs and classrooms in a variety of domains (e.g., programming: Anderson, Conrad, & Corbett, 1989; chemistry: Johnson & Holder, 2010; algebra: Koedinger, Anderson, Hadley, & Mark, 1997; Ritter, Kulikowich, Lei, McGuire & Morgan, 2007; physics: VanLehn, et al., 2005), they are still relatively rare. The one exception is the Carnegie Learning Cognitive Tutor for mathematics, which has been used by hundreds of thousands of students in the United States. Part of the issue, if not the main issue, of why model-tracing tutors are not more prevalent is due to the high barrier of entry in creating such tutors. An intelligent tutor, model-tracing or otherwise, requires at minimum a student interface, a curriculum, and an expert model for checking student answers (the cognitive model for a model-tracing tutor). Creating these pieces requires knowledge of computer programming, pedagogy, and cognitive science. No one person typically has all these skills, so groups of people tend to work on model-tracing tutors. That puts the creation of such tutors solely in the realm of research labs and dedicated companies. We created xPST to alleviate these issues, and to enable individual designers and instructors to create intelligent tutors that behave like model-tracing tutors. That is, it will monitor student response on a step-by-step basis, ensuring they stay on path. xPST works with existing interfaces (e.g., existing websites and off-the-shelf software, including such things as the Torque 3D game engine) so that a custom interface does not need to be programmed. xPST can be extended to work with such third-party software by writing a plug-in that communicates between the interface and the xPST Engine. xPST does not have a formal cognitive model that needs to be created in order to tutor in a domain. Instead, authors create instruction that is problem-specific. This lack of a cognitive model bypasses the need for cognitive science knowledge, such as knowledge representation and production rules. The xPST tutor is then not as powerful as a model-tracing cognitive tutor, but for the specific problem, the xPST tutor can offer similar feedback and interactions as a model-tracing tutor, i.e., it checks each student input, and provides help and just-in-time messages. However, unlike an ITS with a true cognitive model, it cannot generalize to novel problems. We turn now to a brief discussion of related work, including xPST's lineage, and other authoring tools for intelligent tutors. The next sections then describe xPST's architecture and the different implementations we have done, followed by a discussion of the various evaluations done of xPST.

RELATED WORK

xPST can trace its history back to one of the earlier and more successful model-tracing tutors, the programming and algebra tutors developed by Anderson and his colleagues (Anderson, Corbett, Koedinger, & Pelletier, 1995). These model-tracing tutors were built to test Anderson's ACT theory of cognition (Anderson, 1993). To construct these tutors, the Tutor Development Kit (TDK) was programmed in LISP (Anderson & Pelletier, 1991). The TDK borrowed heavily from the ACT model, using production rules to represent procedural knowledge and schematic structures to represent declarative knowledge. Coupled with needing to know these higher-level cognitive science representations, authors also needed to know LISP in order to use the TDK. However, the type of interaction generated by the TDK tutors is reflected in how xPST tutors interact with learners.

In order to more readily deliver commercial-quality tutors, the authoring and delivery systems were separated. The tutor delivery environment, the Tutor Runtime Engine (TRE), operated on a different representation than what the authors used. Ritter, Blessing, and Wheeler (2003) discuss the authoring representation. The TRE tutors were equivalent to TDK tutors and still required programming in LISP at authoring time. At run-time the cognitive model was represented as a state machine that provided the tutor's responses to the student's input. This implementation allowed for a clear separation between the student's interface and the cognitive model. This separation becomes quite useful for xPST and its ability to tutor using third-party interfaces.

Because the learning curve was still high for creating TRE tutors, Carnegie Learning made the Cognitive Tutor SDK (Blessing, Gilbert, Ourada, & Ritter, 2009). In this SDK authors create declarative structures as an ontology, and procedural knowledge as a predicate tree. This tree contains the hints and just-in-time error messages that can be presented to the student. This permits a representation of the cognitive model that allows the SDK to create the tutoring needed for a variety of problems. The SDK continues to be the development tool to create their Cognitive Tutors, and allows non-cognitive scientists to create, modify, and debug their tutors. However, the SDK is designed to create large, robust tutors, usually with enough instruction for several weeks. There is still a considerable learning curve for it, and it is a proprietary tool. Our desire to create something even more straightforward for people to learn and use led us to create the xPST authoring tool.

CTAT

The previous section described the lineage of xPST. Another authoring tool shares that same lineage, the Cognitive Tutor Authoring Tool (CTAT; Aleven, McLaren, Sewall, & Koedinger, 2009). Authors can create two types of cognitive models with CTAT: 1) example-based tutors, and 2) cognitive tutors. Example-based tutors use a programming by demonstration technique. The author demonstrates the solution to a particular problem using the interface the student will use, providing help and other messages during the process. The end result is a tutor for that specific problem, though there are some provisions made for generalization. A cognitive tutor is more similar to what would have been created in the TDK. These two types of tutors represent the two end points between ease-of-use and expressiveness: example-based tutors are easy to make, but not as general, whereas cognitive tutors can be more general, but are harder to create. xPST creates tutors akin to the example-based tutors. A comparison of the advantages of CTAT's graphic user interface vs. xPST's text-based user interface is provided by Devasani, Gilbert & Blessing (2012) and described in more detail later.

Demonstr8

Other systems have also attempted to make the authoring of cognitive models for model-tracing systems, and other types of tutors, simpler. Demonstr8 (Blessing, 2003) allows a domain expert to create via programming by demonstration a model-tracing tutor in a short time with the full capabilities implied by it: a production system that monitors the student’s progress, giving feedback when requested or necessary. That early work inspired us to examine other programming by demonstration solutions, but without the high domain-specific knowledge that was embedded within Demonstr8. That embedded knowledge made tutor creation easier for the author, but limited the scope of the authoring tool. Others have also sought more domain-independent solutions, using other types of tutors, while reducing the cognitive science and programming knowledge needed by the author. For example, Munro (2003) and Towne (2003) have created tools for developing tutors based on device simulations.

Constraint-Based Tutors

Other researchers have developed authoring tools for other types of tutors. Mitrovic and her colleagues (2009) discuss ASPIRE, a system to develop constraint-based tutors. This system also is consistent with our two goals: to allow non-programmers and non-cognitive scientists to author ITSs, and also to use existing interfaces as the student interface. ASPIRE was preceded by WETAS (Martin & Mitrovic, 2002), and added the capability of creating a domain model that could be authored and deployed in a web-based system. Being constraint-based tutors, both WETAS and ASPIRE use syntactic and semantic constraints, as opposed to the rules of a model-tracing tutor, in order to verify student solutions. Constraints specify features of correct solutions, instead of a path through the solution space as in a rule-based system. Constraint-based tutors can more easily handle cases where there are multiple solutions or multiple solution paths, as the tutor checks to make sure constraints are followed and not for a specific path via rules. However, like rules, constraints can be difficult for non-programmers and non-cognitive scientists to grasp, so ASPIRE uses machine-learning algorithms in order to generate the initial set of constraints. In one test of automatically creating constraints, the algorithm used within ASPIRE produced 90% of the needed constraints (Mitrovic et al., 2009). Also consistent with our aims, its designers have used ASPIRE to embed a constraint-based modeling tutor into an existing piece of software (DM-Tutor; Amalathas, Mitrovic, & Ravan, 2012). ASPIRE generated constraints that were used within a live system, one that tutored on managing oil palm plantations.

Ontological Engineering

As a last example of other types of tutors for which researchers have built authoring systems, ontology engineering has also been explored (e.g., Hayashi, Bourdeau, & Mizoguchi, 2009). Most, if not all, tutors require a robust declarative representation to support the tutoring process. Hayashi and colleagues presented OMNIBUS, an ontology that attempts to cover different theories of learning. Its database defined over 1,100 concepts. SMARTIES is built on top of OMNIBUS, and provides a standards-compliant authoring tool to create instruction using the OMNIBUS database. The three-fold goals of the project were to create a system that a) understood a variety of learning theories; b) used those theories in the creation of tutors; and c) allowed sharing of those tutors. One of the advantages of these tutors is that they are not tied to particular instructional models. This focus on instructional strategies and providing alternative learning models differentiates this approach from the authoring systems discussed above.

xPST builds upon the lessons learned from the previous systems discussed in the above paragraphs. Like the example-tracing tutors of CTAT, it makes tutor creation straightforward by only requiring the user to create a “cognitive model” of a single problem. In such a way, it’s not truly a cognitive model, because there’s little to no abstraction over instances or true generalizability of the encoded knowledge. This simplifies the creation of the code that allows the model-tracing-like behavior. ASPIRE uses its machine learning algorithms to attempt to induce the constraints, and xPST has nothing similar. Again, while this limits the generalizability of the authored tutors in xPST, we feel such an approach simplifies the authoring process, making it accessible to a wider range of authors. The next section discusses the xPST architecture.

XPST ARCHITECTURE

As we have mentioned, being able to use an existing interface with a tutor reduces both time required to develop the tutor and any issues of learning transfer. The xPST authoring system was developed to allow tutoring on any interface. It also allows an author to quickly create a model for a particular problem instance by creating hints and other tutoring aspects while the author manipulates the interface. The power of this system is to enable the creation of a model-tracing-like tutor to be built on any piece of software, where it maintains the original interface of the application. The code for xPST is available at http://code.google.com/p/xpst/.

TutorLink makes xPST extensible (see Figure 1), in a manner similar to what it did for the TRE (Blessing, Gilbert, Ourada & Ritter, 2009; Blessing, Gilbert, Blankenship, & Sanghvi, 2009). Its plug-in architecture was inspired by Ritter and Koedinger (1996), and Cheikes et al. (1999) describe a similar architecture. It serves as the intermediary between the third-party application and the xPST Engine. It knows how to map actions in the interface to the proper pieces in the tutor model and how to display hints and other tutoring information within the application. For each third-party application with which the xPST Engine needs to interact, a TutorLink plug-in, or driver, needs to be written that mediates the tutoring interaction between xPST and the application. We have written plug-ins for Microsoft's .NET framework, the Firefox web browser, and the Torque Game Engine. While we did not time our developers as they built these plugins, they estimate approximately a week of full-time work for each plugin for an experienced programmer, and somewhat longer for a graduate student. We will discuss their details later in the implementation section.

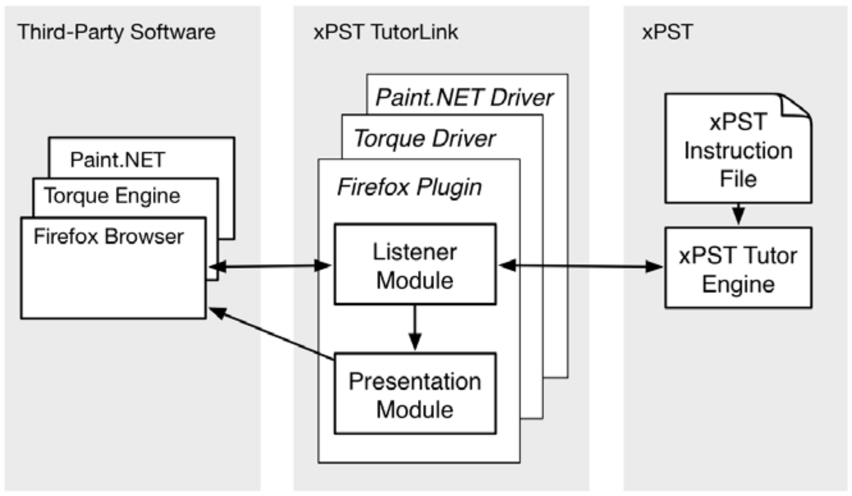


Figure 1. xPST Architecture

Even though writing a driver for TutorLink takes expertise, it only needs to be done once for each new interface. These TutorLink drivers contain the Listener Module and Presentation Module. The Listener Module knows how to map interface widgets in the application to learning objects as defined by the author in the xPST File. We often refer to these learning objects as goalnodes, as they represent subgoals that the problem solver needs to complete in order to be done with the problem. The Listener Module eavesdrops on user's actions and sends them back to xPST Tutor Engine. The xPST Tutor Engine checks the action with the rules specified by the author in the xPST Instruction File and responds with the appropriate feedback (e.g., allow the action, mark the action as incorrect, display a just-in-time message, or display a help message). The Listener Module takes this feedback and sends a message to the Presentation Module, which then determines the proper way to display it given the widget. For example, text fields will need to display the feedback differently than pop-up menus. The xPST Tutor Engine runs on its own server or locally, and communicates with other components via TCP/IP, which allows the tutored application and tutor to be separated on different servers if need be.

As seen in Figure 1, the xPST Tutor Engine operates on an xPST Instruction File, which is a text file designed to be easy to read and write by non-cognitive scientists and non-programmers. It contains three sections: Mappings, Sequence, and Feedback. The Mappings section serves the purpose of connecting inputs from specific widgets or ports in the third-party software to the goalnodes. In Table 1, how would xPST know that the resolution has been changed? If TextBox_7452 is the entry box for changing the resolution, then once that box is mapped, the Listener Module will send any learner inputs into that box to the ChangeResolution goalnode. Mapping goes in both directions, so that if the ChangeResolution goalnode detects an error, then the Presentation Module for this tutor might make

TextBox_7452 turn red to indicate an error. The syntax for the Mappings section is a list of tuples, the name of the widget as it comes over from TutorLink, and then an author-defined name.

In the Sequence section, the tutor author lists subgoals, or goalnodes, that the learner should accomplish. In the example in Table 1, a learner needs to first change the resolution of an image, and then click Okay. The syntax for this section can be enhanced from what is shown by the use of Booleans and parentheses. Goalnodes can be separated by the reserved words `then`, `or`, `and`, and `until`. Goalnodes can be completed in order (`A then B`), or there may be optional paths, e.g., `(A then B) or (C then D)`. Sometimes groups of goalnodes must be completed before continuing, e.g., `(A and B) then C`. Finally, under circumstances when goalnodes may be completed, but remain open for changes until a later goalnode is complete (e.g. textboxes in a modal preferences dialog remain editable until the Save button is clicked), the Sequence could be `(A and B) until C`. The author can also indicate if the goalnode is required to be completed or merely optional.

Finally, the tutor needs to know how to respond to inputs for each goalnode. The Feedback section provides hints and error messages for each goalnode within the Sequence and also indicates what the correct answer should be. The syntax for the Feedback section is no more complicated than what is shown. Each goalnode is listed, with its answer, hint sequence, and any just-in-time messages. The answer can be a number or string, or some goalnodes require a different format (e.g., for the image-editing tutor shown in Figure 2, we had goalnodes that needed to respond to RGB values; such functionality is added within the TutorLink module). The hints and just-in-time messages are simple strings. The just-in-time message requires a simple Boolean argument that indicates when they are triggered (e.g., in the example shown, when the user types “300”). Just-in-time messages are processed in order, such that if multiple ones match, only the first one is shown.

Because of the easy-to-understand syntax of the xPST file, a text editor is sufficient to create the file. Table 1 shows the beginnings of a xPST file, with two interface elements mapped to goalnodes, the sequence of those goalnodes, and the feedback associated with one of those goalnodes. The syntax has programming elements, but the code does not get more complicated than what is shown. As our results demonstrate (see later sections), even non-programmers when given a template quickly master what syntax there is.

Table 1. A sample xPST File

```
mappings
{    TextBox_7452 => ChangeResolution;
     Button_3461 => OkayButton;   }

sequence
{ ChangeResolution then OkayButton }

feedback
{
     ChangeResolution {
          answer: 72;
          Hint: “You need to input a resolution for a webpage image.”;
          Hint: “The best resolution for your purpose is 72 dpi.”
          JIT {v == “300”}: “You entered 300, which is for printing.”; }
}
```

The most challenging aspect of the xPST system is creating the TutorLink plug-ins that handle the communication between the third-party application and the xPST Engine. We have used three different ways in order to listen for user events within the third-party software. First, we have either added widgets to open source software or created widgets using the third-party software's API to send and receive the needed actions (cf. Ritter & Koedinger, 1996). Second, for software that has little API but does have accessibility hooks designed to be used by screen reader software like JAWS or NVDA, the Listener Module can use those cues to know what the user is doing. Screen recording software such as Adobe Captivate or TechSmith Camtasia use a similar approach. Third, for software that has neither an API nor accessibility hooks, we can use low-level OS events, such as window focus and mouse events. This approach is the most difficult, because inferring what a user is doing from low-level events can be complex. Note that while the implementations described below have all used desktop PCs, the architecture is designed to support tutoring on mobile devices.

Another aspect that makes xPST extensible is the ability to extend the types of learner inputs that the xPST Engine can check. We refer to these as “checktypes.” It has common checktypes built-in, such as comparing strings and numbers. Different interfaces may need the ability to check different types of student answers beyond these built-in types. For instance, when checking student answers in an image manipulation program, it may be necessary to compare a learner's choice from a color picker dialog with an allowable range defined by the tutor author as “light blue.” For this example, a checktype can be added to xPST to handle value-triples (either Red, Green, and Blue values, or Hue, Saturation, and Brightness values). This modular architecture allowed us to implement ConceptGrid (Blessing, Devasani, & Gilbert, 2012; Devasani, Aist, Blessing, & Gilbert, 2011), which is a collection of checktypes to compare short natural language sentences within xPST. In one of the implementation examples below, we also describe how we added the capability to monitor and check student progress in a 3D game environment.

Challenges in Tutoring on Third-Party Software

Across the various implementations of xPST tutors that we will discuss below, we have experienced several challenges in how to develop tutors using third-party software as the learner’s interface (Gilbert, Blessing, & Blankenship, 2009). We will discuss three design challenges: 1) integrating a problem scenario and instructional feedback into the context of the existing application, 2) providing learners with appropriate feedback during or after task performance, and 3) allowing learners to explore the interface while also making sure they complete the task.

The first challenge is related to the idea of embedded training, i.e., conducting training on a system within the same context as the live system itself, rather than in a separate application. In our context, the Presentation Module needs the capability to display the problem scenario to the learner. In the tutors we have produced, we have enabled either a “Tutor” menu in the menu bar or we added that option to an existing menu. Learners can initiate the tutor via this menu. The problem statement and other information are then displayed in either a sidebar or a floating window. One aspect of this challenge is ensuring that the learner is able to distinguish between the system being tutored on and the training components that will later disappear when the educational scaffolding is removed. This challenge can also be addressed by having the tutor components display a more casual look and feel than the rest of the interface, for example by using bright color and hand-drawn-looking coachmarks. Similarly, Kumaraguru et al. (2007) found a comic-strip approach to be effective, and Kang, Plaisant and Shneiderman (2003) used a sticky note metaphor to provide “integrated initial guidance” for a task.

Indeed, the Presentation Modules that we have written for our tutors use those techniques, and they can be observed in the screenshots of xPST tutor presented in later sections.

To address the second challenge, providing learners with appropriate feedback, hint and just-in-time messages need to be displayed to the learner. Messages can be displayed to the learner in either a modal state, which requires immediate attention by the learner before proceeding (e.g., "Click Okay to continue"), or non-modal (e.g., a sidebar alert), which the learner may choose to ignore. This issue is of particular interest when learners make an incorrect action. In order for the learner to see the consequence of the action, the tutor could allow the interface to perform the action, but then put up a modal dialog that would force an undo in order to get the user back on the solution path. Research from the original ACT tutors (Anderson, Conrad, & Corbett, 1989) suggests that having learners stay on path is more useful than allowing learners to explore fruitless paths. Also, when building a tutor on top of an existing interface, the tutor must instruct the learner about the interface itself in addition to the domain knowledge relevant to the task. The tutor must therefore provide guidance to help users overcome any usability flaws in the underlying system, over which the tutor author has no control.

One concern with providing such feedback overlaid on an existing interface is that the tutored instruction may obscure important parts of the interface. The indicators for right and wrong answers are simple highlights to the widget, and so we have not experienced problems in that regard. In all cases we have made the windows that provide hints and just and time messages movable. In such a way, if the default positioning of the window obscures a needed part of the interface, the learner may simply move the window.

The third and final challenge to discuss here relates to allowing the learner the freedom to explore the problem space while still guiding them to solution. In a traditional ITS, where the interface was built especially for the tutor, blocking an action as discussed above usually does not occur. The ITS authors constructed the interface to streamline the responses by the student. When using an existing interface the ITS author needs to consider all the possible actions that a learner might take in that interface. The author also must consider that different actions might lead to the same state (e.g., using a keyboard shortcut instead of a menu selection). A trade-off exists between the amount of flexibility allowed and the complexity needed within the tutor. More flexibility generally equates to more complicated tutors. We attempted to follow a "middle-of-the-road" approach in which learners are granted flexibility to explore, but guided back on path as quickly as possible, blocking them if necessary. For example, in the Paint.NET tutor described below, we allowed users to explore the drop menus of the software application 7-10 times (depending on how you count sub-menus), because that exploration is a natural part of learning. Also, we planned feedback for menu choices that suggested reasonable errors, i.e., they were not the correct choice but were related to the correct choice. However, if users chose menu items that were unrelated to the task, we actually blocked those menu items from being activated and gave a message reminding the user of the current goal.

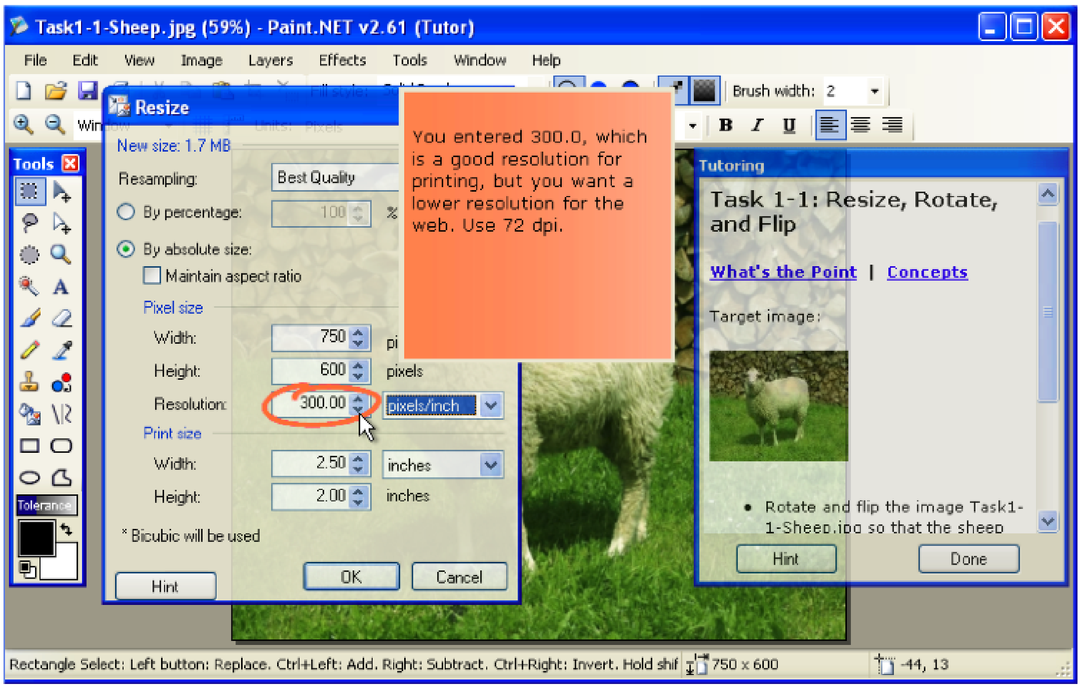
THREE TUTORLINK IMPLEMENTATIONS

We turn now to short discussions of specific implementations of xPST that we have produced. These correspond to three different TutorLink plug-in modules we have developed that allow the xPST Tutoring Engine to tutor on third-party software. In discussing the implementations in this section we will briefly mention some of the effectiveness data we gathered of learners using the content. The section following this one discusses our findings concerning the usability of the xPST authoring tool itself.

Microsoft's .NET Framework: Paint.NET Tutor

Our first xPST-based tutor that had the TutorLink protocol to link the xPST Engine to a third-party piece of software used Paint.NET as that third-party software. This is a re-instrumented version of the tutor discussed by Hategekimana, Gilbert, and Blessing (2008), originally done using the Cognitive Tutor SDK to provide the tutoring. Paint.NET is open-source photo editing software that runs under Microsoft's .NET framework. Because Paint.NET is open-source, we could use the first method of plug-in development described above: we added ITS windows and menu options directly into the client. Software developers integrated the ITS instruction within the Paint.NET application itself, allowing the user interface to work seamlessly with the tutor. This is an ideal approach in terms of learning transfer, since the tutoring can be done directly in Paint.NET, the target of the training itself. Thus, the tutoring team does not need to build an interface that mimics an existing piece of software. They can directly use that piece of software.

Figure 2 shows a screenshot of the xPST Paint.NET tutor in action. The learner has been given the task to rotate and then change the resolution of an image to get it ready for the web. The problem statement plus additional information is given in the "Tutoring" window. The learner has made a common error in changing the resolution, and the tutor has circled the issue on the Paint.NET interface and displayed a just-in-time error message to the student concerning the issue. For this to work, the TutorLink .NET plug-in communicates to the xPST Tutor Engine what widget has been manipulated in the interface. The xPST engine checks to see if that widget is contained within the xPST Mapping listing. If so, the Engine will map the widget name to the goalnode name. The xPST Engine will then check to see if it's an appropriate time to perform an action on that goalnode via the Sequence. If so, it then checks its Feedback rules. Depending on that result, the xPST engine will communicate with the Presentation Manager, which will produce the proper display for the student.


Task1-1-Sheep.jpg (59%) - Paint.NET v2.61 (Tutor)
File Edit View Image Layers Effects Tools Window Help
Brush width: 2
Tools
Tolerance
Resize
Hint
OK
Cancel
Tutoring
Hint
Done

New size: 1.7 MB
Resampling: Best Quality
By percentage: 100 %
By absolute size:
Maintain aspect ratio
Pixel size
Width: 750 pixels
Height: 600 pixels
Resolution: 300.00 pixels/inch
Print size
Width: 2.50 inches
Height: 2.00 inches
* Bicubic will be used

You entered 300.0, which is a good resolution for printing, but you want a lower resolution for the web. Use 72 dpi.

Task 1-1: Resize, Rotate, and Flip

What's the Point | Concepts

Target image:

- Rotate and flip the image Task1-1-Sheep.jpg so that the sheep

Rectangle Select: Left button: Replace. Ctrl+Left: Add. Right: Subtract. Ctrl+Right: Invert. Hold shif 750 x 600 -44, 13

Figure 2. Screenshot of the Paint.NET xPST Tutor.

The efficacy of the instruction given by the Paint.NET ITS was tested by 75 students doing the same exercises either by learning from a book excerpt or by watching an expert perform the steps in a screen capture movie. A couple students didn't complete one or more of the dependent measures; thus the sample sizes vary slightly by measure, are typically 24 Book, 24 Movie, and 26 ITS. After doing 12 learning exercises in one of these training methods, participants' ability to answer questions and do transfer tasks was assessed. The groups' concept acquisition was not statistically different (Book *M*=6.9, *SD*=1.7; Movie *M*=6.1, *SD*=2.2; ITS *M*=6.1, *SD*=1.7; *F*(2,72)=1.459, *p*=.239). Task performance was significantly different among the groups, with a medium effect size (Book *M*=25.0, *SD*=8.8; Movie *M*=17.7, *SD*=9.4; ITS *M*=22.3, *SD*=8.1; *F*(2,64)=3.859, *p*=.026, ω^2=0.08). Tukey post-hoc analysis revealed that the mean task performance of the book group was significantly higher than the movie group (7.3, 95% CI [0.78, 13.82], *p*=.025), but other group differences were not significant. Unfortunately even though the concept acquisition scores were not significantly different, on a system usability scale (SUS) adapted from Brooke (1996), the student's perception of the ITS's usability was significantly less than that of the book and movie (Book *M*=60.3, *SD*=10.6; Movie *M*=60.0, *SD*=9.0; ITS *M*=50.9, *SD*=8.7; *F*(2,71)=7.611, *p*=.001, ω^2=0.15). From the student's feedback, the students in the ITS condition thought the system did not provide enough feedback and allowed too much freedom to explore the problem space within this authentic application. We took these comments in consideration when we constructed the next xPST tutor.

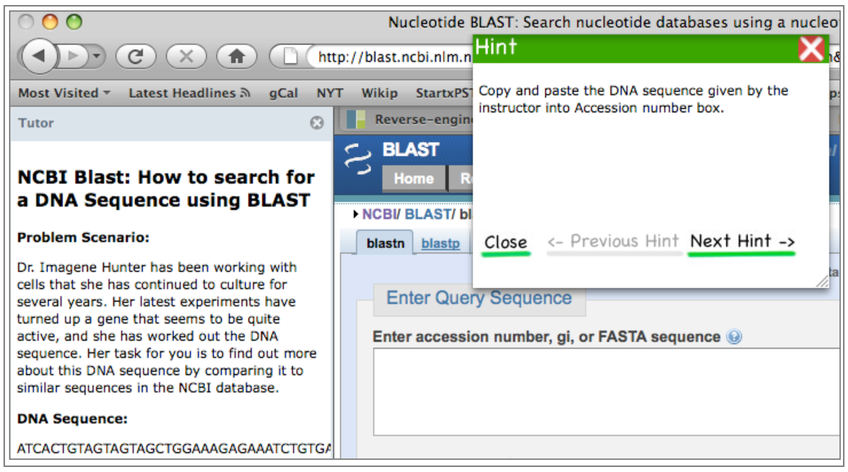
Firefox Web Browser

Many computer-based interactions are now handled via web browsers. In order to tutor learners where the content occurs on the world-wide web, we created a TutorLink plug-in that works in the Firefox web browser that allows the xPST Engine to interact with the widgets contained within a browser window. The creation of this plug-in occurred in two phases. Our initial attempt was conducted in conjunction with CAPE (Courseware Authoring and Packaging Environment), developed by the VaNTH ERC (Roselli, et al., 2008). CAPE itself is an authoring tool, used to create online assignments for quantitative engineering problems. The CAPE creators wanted to design an ITS to instruct its beginning users. To that end, we developed a Firefox TutorLink plug-in that allowed the xPST Engine to communicate with the widgets used on the CAPE website. Instruction for a two-hour portion of the CAPE workshop was developed using xPST, consisting of four problems of increasing complexity and length. This plug-in was also developed using the first method of plug-in development described above, as Firefox has a plug-in API that is available to the public.

A comparison was made between attendees at a pre-ITS workshop and attendees at a workshop where the CAPE instruction used the ITS. The ITS users completed the exercises in 14% less time and reported less frustration at learning to use CAPE. However, they also reported that the system should be less restrictive in how it expected users to navigate the system, in contrast to our experiences with the Paint.NET students. We also experienced a few technical issues with regards to synchronizing the monitoring of the student interface and providing feedback to the student.

The second phase of the Firefox plug-in attempted to address these issues, making the tool more usable and adaptable (Blessing, Devasani, & Gilbert, 2011). The end result is what we refer to as Web xPST. Web xPST can be accessed at http://xpst.vrac.iastate.edu. Using the website, authors sign into an account and create xPST-based tutors for web-based content. Web xPST has an embedded text editor that allows the creation and saving of xPST Instruction Files directly to the web. When a learner navigates to a site that has an associated xPST tutor, the xPST Engine provides the tutoring on the content. The Firefox plug-in shows the feedback to the student and contains a side pane through which the author can provide the learner with instructions concerning the problem scenario, and a toolbar through which the learner can ask for help and indicate they are done solving the problem (see Figure 4). The hint window, the red X, and the green check are all feedback generated by the xPST Engine that the Presentation Module displayed in the browser. The common web widgets can all be accommodated, and others can be added by extending the Listener Module and the checktypes of the xPST Engine. Existing websites contain much content that could be tutored, and this capability also opens up the possibility of an instructional designer creating their own problems and instruction on a website, and using Web xPST to create a model-tracing style ITS. Many tools exist to allow the easy creation of interactive websites.

We initially demonstrated Web xPST's ability to produce tutors for third-party websites by creating a tutor in collaboration with an AP Biology teacher. This tutor guided high-school students in using the NIH NCBI Bioinformatics tools that are available online. The students were given homework problems in DNA sequencing in a left-hand problem pane, and the online websites were available in the right pane (Figure 3). This provided a proof-of-concept for Web xPST.



Copy and paste the DNA sequence given by the instructor into Accession number box.

NCBI Blast: How to search for a DNA Sequence using BLAST

Problem Scenario:

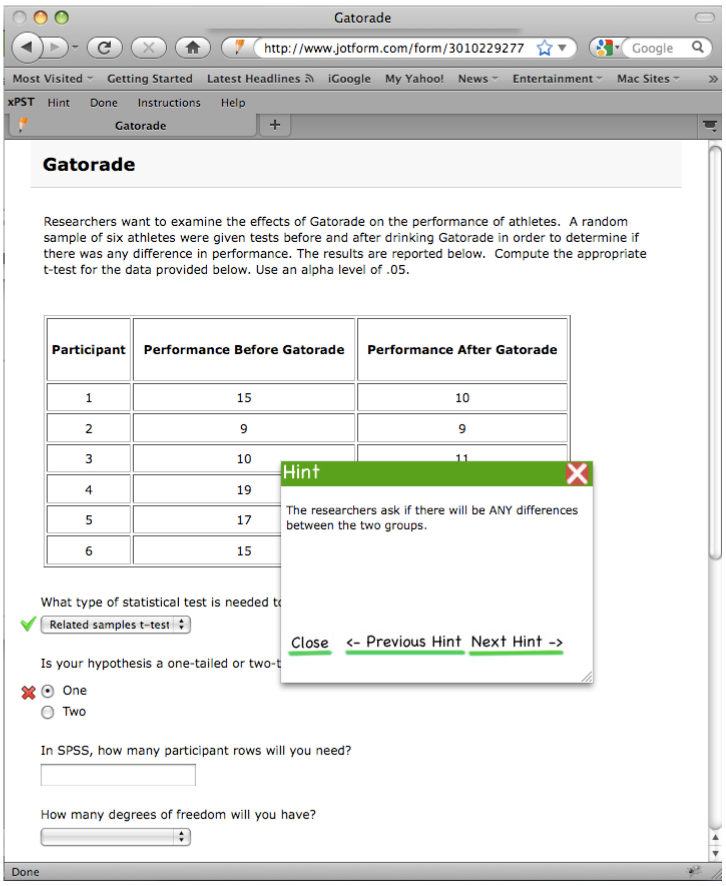
Dr. Imagene Hunter has been working with cells that she has continued to culture for several years. Her latest experiments have turned up a gene that seems to be quite active, and she has worked out the DNA sequence. Her task for you is to find out more about this DNA sequence by comparing it to similar sequences in the NCBI database.

DNA Sequence:

ATCACTGTAGTAGTAGCTGGAAAGAGAAATCTGTGA

Figure 3. Example of using an xPST tutor on a third-party website, the NIH's BLAST genetic sequence search tool.

We used a custom content approach when we created a statistics tutor, called xSTAT, using Web xPST (Maass & Blessing, 2011). The authors used JotForm, a simple web form creation tool, to create the problems (see Figure 4). This project provided a more strenuous test of the effectiveness of xPST-created tutors. To test the tutor's effectiveness, students were randomly assigned to do a homework assignment either using the ITS-enabled version of the web problems (n=24), or to do the same problems on the web but without the ITS-based feedback (n=25). The problems were all very involved, multi-step ones that used real-world data sets. At the time of the final exam, all students solved two transfer problems. The non-ITS condition averaged 62% (SD = 6.99) and the students with tutored instruction averaged 74% (SD = 4.27) on these problems, a significant difference ($t(47) = 2.03$, $p < .05$, $d = 0.62$). A Mann-Whitney test indicated that the tutored group had higher satisfaction (Mdn = 4) than the non-tutored group (Mdn = 3), $U = 163.50$, $p < .05$, $r = .32$.



Gatorade

Researchers want to examine the effects of Gatorade on the performance of athletes. A random sample of six athletes were given tests before and after drinking Gatorade in order to determine if there was any difference in performance. The results are reported below. Compute the appropriate t-test for the data provided below. Use an alpha level of .05.

Participant	Performance Before Gatorade	Performance After Gatorade
1	15	10
2	9	9
3	10	11
4	19	
5	17	
6	15	

Hint

The researchers ask if there will be ANY differences between the two groups.

Close <- Previous Hint Next Hint ->

What type of statistical test is needed t

✓ Related samples t-test

Is your hypothesis a one-tailed or two-t

✗ [✓ One] Two

In SPSS, how many participant rows will you need?

How many degrees of freedom will you have?

Figure 4. A problem in the xSTAT Web xPST system.

Game Engines

We extended xPST to enable authoring tutors in 3D games (Devasani, Gilbert, Shetty, Ramaswami, & Blessing, 2011; Gilbert, Devasani, Kodavali, & Blessing, 2011; Kodavali, Gilbert, & Blessing, 2010). The military has made great use of such 3D game environments in order to provide personnel training in a variety of different scenarios. Our goal is to provide military trainers, as well as others who author scenarios in 3D environments, with the ability to create in-scenario tutoring in an appropriate and easy-to-author manner. Livak, Heffernan, and Moyer (2004) also created a tutor for a military 3D game environment, using JESS for the feedback rules. Johnson and Valente (2009) used related AI techniques for language instruction in a virtual environment using Alelo. Our goal is to extend the benefits of the xPST architecture to this task of tutoring in a virtual environment, using simpler authoring of rules and a tutor structure that is independent of the game engine chosen. Authoring in this context involves these issues: 1) navigating a 3D environment is quite a bit different than navigating a 2D software interface; 2) complex domain knowledge is required to play and navigate in the scenario; and 3) different types of feedback and interactions are generated by various entities apart from the learner. Research has shown that students learn better and retain more when they actively engage in the learning process (Bell, Lewenstein, Shouse, & Feder, 2009). Researchers have also found face-to-face interaction with animated pedagogical agents provides greater promise in interactive learning environments (e.g., Johnson, Rickel, & Lester, 2000; Kim et al., 2009; Leelawong & Biswas, 2008). Tutoring using games potentially provides these advantages more so when compared to tutoring with traditional software. Pedagogy researchers have shown an increased interest in incorporating gaming principles into teaching and learning (Gee, 2005). Games manage to maintain the user's attention with a background story, high-end graphics and the feeling of immersion within a simulated environment (Shute, Ventura, Bauer, & Zapata-Rivera, 2009). Most of the ITS authoring tools focus on creating ITSs for static interfaces, such as a radio buttons and text boxes. Few, if any, ITS authoring tools exist for developing tutors for game engines like Unity or other synthetic environments.

It is worth noting the conceptual differences between tutoring in a typical graphic learner interface versus in a complex fast-moving game. In a GUI, a learner’s action will typically evoke similar responses if done repetitively. On the other hand, in the synthetic environment of a 3D game, a learner’s actions are frequently dependent on other entities and the time course that has transpired. In a synthetic environment, a learner’s goal is not to complete a textbox with a certain correct answer but rather to reach a specific game state, such as having defeated all the enemies or reached a particular location. Some of the details of the learner's path to reach that game state may be irrelevant to the learning objective.

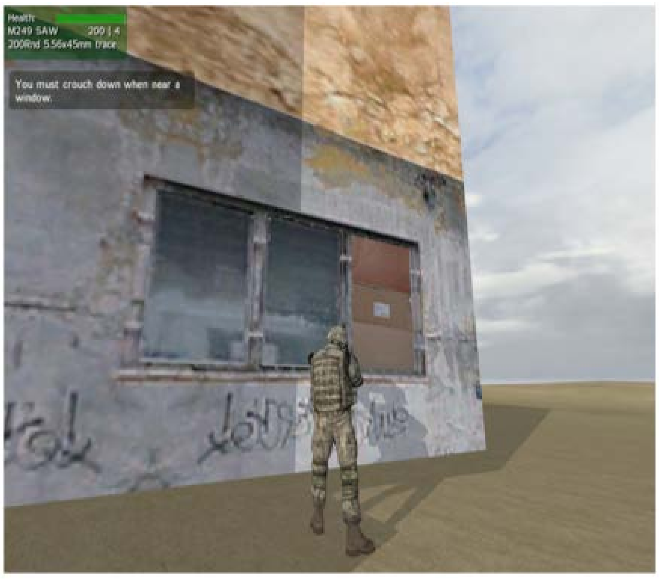
Changing the application’s state within the game state space is a main focus in synthetic environments. The granularity at which these states need to be defined depends on the author and the complexity of the task. Unlike the traditional GUI software or website, synthetic environments are a dynamic system where interaction can happen between various entities apart from the player, and events can be triggered by different entities. Thus, it is useful to categorize the events as player events or non-player events. Also in a 3D game, learners are required to navigate through a simulated environment and sometimes communicate with other entities. This calls for the tutor authoring system to provide tools to support these actions. Furthermore, in addition to hints and just-in-time error messages, feedback from the synthetic environment of a 3D game tutoring system has a third type, prompts. These are neither requested nor based on an incorrect event. Instead, they convey important messages such as a certain subgoal has been completed or a danger warning. These can be time-based or non-player object based. In order to accommodate the differences between 3D games and traditional GUI software or websites, we used the extendable nature of the xPST Engine to add the following four extensions to the xPST architecture to facilitate easy authoring of ITSs in 3D games.

Actions by non-player objects. Events can be triggered by non-player objects in 3D games. These can include such actions as a bomb exploding at a particular time point during the game or an enemy soldier firing on the player. In order to accommodate these non-player initiated events, we devised a more generic way of handling events compared to the previous xPST architecture in which the unique ID attribute always corresponded to the player object.

Proactive hints or prompts. Because of the complexity of 3D games and multiple interactions between entities, it is sometimes not obvious what the current game state is. It becomes useful for the learner to have direct feedback when the current goalnode is completed or to receive some reminders about the next goalnode. We have included a new type of feedback in xPST, “OnComplete,” supplementing the potential hints and just-in-time messages for each goalnode. This feedback is proactively provided to the student on completion of that particular goalnode.

Communication events. Many of the tasks in 3D games require the student to communicate with other player entities in the game. We extended xPST to support tutoring on communication events by using a special goalnode (“StartTalk”) to initiate the communication with other entities. The student is able to choose the entity with which to communicate and the message to communicate.

Location events. Location events facilitate tutoring on the navigational aspects of the player's performance. Unlike traditional GUI software, almost every task in a 3D game requires the player to move within the synthetic environment. We provide a particular goalnode type to use when the player enters a particular designated location or geographic region within the game. Figure 5 shows a screenshot of a Location Event being used.



You must crouch down when near a window.

Figure 5. Example of a Location Event. The learner receives a prompt (upper left) reminding, "You must crouch down when near a window."

We implemented a TutorLink plug-in to allow the xPST engine to communicate with Torque Game Engine Advanced (TGEA), which was used as our simulation engine. It is a commercial off-the-shelf engine that provides various core functionalities for game development. It reduces game development time and allows the author to concentrate more on the tutor development. We created the xPST Torque plug-in using TorqueScript, part of Torque's public API. Thus, this plug-in was developed using the first of the three methods of plug-in development described above. The Listener Module listens to the events in the game and sends them back to the xPST Engine for evaluation. Then the xPST Engine sends the appropriate tutoring feedback to the Presentation Module, which displays this feedback to the user within the game. Similar plug-ins could be created for the other 3D game engines that are in use.

To demonstrate the efficacy of our additions to the xPST architecture to provide tutoring in these synthetic environments, we developed several game scenarios, one of which is called Evacuate. The task teaches the learner how to evacuate civilians from the buildings in the scenario. The scenario has three buildings, each with one civilian inside. The player enters each building, checks for civilians, communicates with the civilians (giving an evacuation command), and waits for the civilians to come out. When the learner does this for all the buildings in the scenario, the task has been successfully completed. In an evaluation described in Gilbert et al. (2011), 10 non-programmers were given the task of developing tutors for two Torque scenarios, first a simpler one, and then the more complex Evacuate one. Each participant was given training in the form of a 15-minute video demonstrating the creation of a simple tutor for Torque. Additional optional materials were made available: a 45-minute detailed video, five online documentation-style webpages, and a sample xPST file with comments. These materials required approximately two hours to view completely. Participants created the two models at their own pace over a two-week period. Participants demonstrated that they could author the tutors using xPST using this small amount of training. Some participants demonstrated a strong learning curve, with their time to author the more complex second scenario being less than half the time to author the simpler scenario. Scenario authoring times varied significantly, from 3 - 35 minutes.

AUTHORING EVALUATIONS

We have conducted three more extensive authoring evaluations of people using xPST to author problems. The first two that we will discuss use two different iterations of the Web xPST implementation. The third study is a comparison between using xPST and CTAT to author various types of tutors.

Initial Study

To explore the ease of authoring an xPST tutor for web-based tasks, we conducted a study in an introductory human-computer interaction graduate class using an early version of the Web xPST (Gilbert, Blessing, & Kodavali, 2009). Ten students, all non-programmers, attempted an extra credit assignment. The students had to design an ITS using xPST to tutor users on how to search using a particular web-based library search engine, the ACM Portal's Advanced Search web form. In order to complete the assignment, the students were first provided with a two-page document containing high-level information on ITSs in general and a 5-minute video demonstrating the xPST CAPE tutor. They were also given an extensive worked example that contained four parts: 1) an annotated xPST example file, 2) a 44-minute screen movie of how to use the xPST authoring interface, 3) a six-page document describing the technical information contained in the movie, and 4) a website where they could try to re-create the worked example themselves. The assignment itself had each student author three individual problem tutors for different types of ACM Portal database search. The three search tasks were designed to be equally complex in terms of tutor authoring, each requiring a similar number of steps. The results showed that the 10 participants produced 26 tutors. To evaluate the tutors, we used the rubric shown in Table 2 to score tutors on a 5-point scale that indicated the quality of the tutor. We used a similar rubric previously in (Blessing & Gilbert, 2008), and other researchers have used similar scales (Martin, Mitrovic, & Suraweera, 2007).

Table 2. How the cognitive models were scored.

Score	Description	Models
5	A model that produces behaviors close to an ideal model, in terms of hints and just-in-time messages	6
4	A very good model that is beyond just being sufficient	12
3	A sufficient model where the student can complete task	7
2	Model provides hints, but does not provide enough guidance for a novice	1
1	Model runs but produces nonsensical help	0

Eighteen (69%) received a score of either 4 or 5, indicating that the tutor went above the minimum needed to scaffold the learner through the task. Seven tutors met the minimum by receiving a score of

3, and only 1 model was deficient in tutoring. Typical errors by model authors included omitting goalnode steps required for the tasks (a less frequent problem, but one that lowered the rating score more heavily) and not providing sufficient instructional support, i.e., leaving out hints or just-in-time messages (a more frequent error that lowered the rating score slightly). This range of quality indicates that not all the models were the same, and that the authors were not solely using the worked example as a template to do their work. The students differentiated their models in their creation, and the three different scenarios required them to do so.

The authoring system tracked the time spent doing the assignment. Not surprisingly, we found that the time on task decreased as the participants moved from task to task: the average time to author Task A was 5.1 hours, for Task B 2.5 hours, and for Task C 1.7 hours. The quickest time was 2.6 hours for all three tutors, and the slowest was 21.0 hours (a non-native English speaker who took 13.6 hours on the first model alone). This timing includes the time the students spent looking at the learning materials. The timing data also tracked which part of the xPST file the participants worked on (mapping, sequence, or feedback). Figure 6 shows what the average participant was doing while authoring the tutor. We analyzed the data for each participant by dividing total authoring time into quintiles. Within each quintile we calculated what percentage of the time was spent authoring sequence code, mapping code, and feedback code. This particular graph is based on all available data (26 tutor models), though all graphs are very similar to one another regardless of how the data are sliced. Much of the sequencing work is done first (in the first quintile of total authoring time), and the mapping and feedback work is then done in tandem. This pattern was shown as the worked example movie went through model creation, and it appears that most of the participants adopted that path.

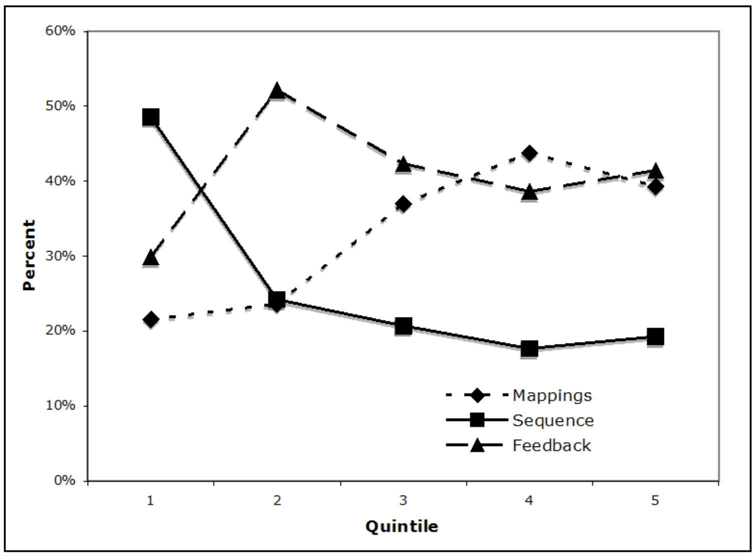


Figure 6. Average authoring activity within Web xPST across all 26 tutors. Participants focus primarily on sequence initially, and then on mappings and feedback.

These problems provide 10 - 15 min worth of training apiece. This initial result provided encouragement to produce a larger set of instructional tutors. We conducted another study, once the Web xPST system had been refined.

A Second Study

As a case study of the final Web xPST system, five non-cognitive scientists and non-programmers developed a set of problems to be used in a college-level statistics course (Blessing, Devasani, & Gilbert, 2011). Two of the authors were instructors of the course, and three were undergraduates who had successfully completed the course in the past. They developed the tutor discussed above under the Web xPST implementation and shown in Figure 4. To develop the materials for this tutor, both the problems and the instruction had to be authored. To support these users, who had no previous HTML experience, they used an online form creation tool (JotForm; www.jotform.com) that allows for a wide variety of widgets to be dragged-and-dropped onto a form for easy layout. The authors then used the WebxPST website to author the instruction and tutoring for the problems they created using JotForm.

We created a set of instructional materials for the participants to learn both JotForm and WebxPST. The JotForm website itself contains a 2-minute video on its use. In addition we wrote a set of instructions that filled less than 1 page. We also created a 7-page WebxPST manual, which included a 3-page example xPST file. Authors had to be instructed on how to map form widgets to xPST goalnodes, e.g. labeling a menu input_18 on the webpage as ChooseTestType. Instructions also described how to sequence these goalnodes in a linear sequence, e.g. A then B then C. Although xPST can accommodate more complicated sequences including branching and alternative solutions, there was no need for that in these simple JotForm webpages. Finally, the instructions described how to indicate the correct answers, hints, and just-in-time messages for these goalnodes. Participants met with the researchers for a 1.5-hour meeting to go over the instructions and see demonstrations of JotForm and WebxPST. This initial learning time is not included in the timing results presented below. Participants had three more one-hour meetings with the researchers over the next month to discuss authoring strategies.

The participants had a goal of authoring 15 problems apiece within one month. All but one of the participants achieved that goal. The one who did not meet the goal authored 12 problems. One author developed 17 problem tutors. These 5 participants authored a total of 74 problems over the course of the month. While the general format of these problems were similar at a high level (a problem setup with several specific questions being asked about the scenario), the actual problems and tutors developed within and between the participants were varied in their questions and content, in a manner similar to the problems found in the back of a statistics textbook chapter, or one of Carnegie Learning’s Cognitive Tutors. We took two time measures of the participants while using the system. One was the total time logged into the system and the other was the time spent just typing xPST code. The total time includes the time to create the form within JotForm and formulate the problem itself. The xPST coding time, included in the total time, is a relatively exact estimate of how long participants spent typing the instructional code (mapping, sequence, hints, and JITs). Figure 7 shows the time course of authoring problems for these participants. Authors averaged 28.57 hr logged on to the system across the month for the total time (with a range of 18.45 to 36.85 hr), and a mean of 7.37 hr editing the xPST Instruction File (a range of 4.87 to 9.48 hr). By the end of the experiment, participants spent less than 45 minutes authoring a problem in total, with just less than 18 minutes of that time spent writing xPST code. Under certain circumstances, e.g., if a goalnode has a similar desired hint sequence in two problems, features of an xPST Instruction File can be copied and pasted between problem files and edited to account for differences in the new problem. Our authors made use of that shortcut. As can be seen, the main effort is not in producing the xPST code itself, but rather in other aspects of authoring, such as coming up with the problem itself. This is what a good authoring tool should do—eliminate the coding burden so that the time can be spent in producing useful instruction.

Each of these problems offered at least 10 min of instruction time, and were the ones shown to be effective in the xSTAT tutoring study discussed above (Maass & Blessing, 2011). All 74 of the models were deemed to meet the minimum needed in order to provide tutoring (using the scoring from Table 2, a 3 or higher). This is borne out by the finding presented previously, that the tutored students learned the material significantly better than the control condition. These results clearly demonstrate the efficacy of this authoring tool to create effective tutors. In addition to these more objective measures, the participant authors filled out two separate attitude surveys across the study comprised of Likert-scale items and free response questions. They ranked Web xPST tutor as powerful and easy to use, and not being frustrated in its use. Participants did mention the "code" aspect of the Web xPST system as an initial hindrance, but after completing their first problem, that issue went away. Indeed, we only fielded eight total support emails across the month, most at the beginning of the month. These were mostly for simple syntactic concerns (e.g., how exactly to format the Boolean for a just-in-time message). The questions that occurred during the meetings concerning authoring strategies were indeed pedagogical in nature, such as how best to structure to a hint sequence for a particular subgoal. Their perception is matched by the timing data of Figure 7, that by the end of the month, they felt most of their time was spent creating the problem itself and devising its hints and JITs, and not on coding. Participants saw the potential value-added for students, and felt that the time it took to learn the system was worth it.

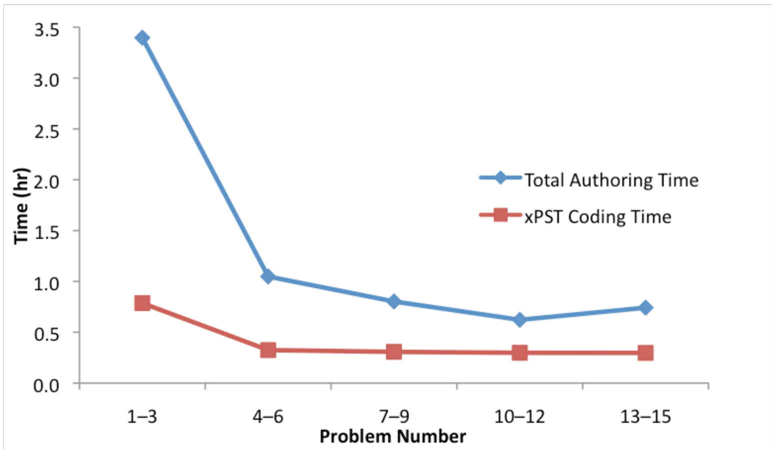


Figure 7. Number of problems authored versus time

Comparison Study

Given their similar lineage but different approaches, we were curious to observe the various merits of xPST with CTAT (Devasani, Gilbert, & Blessing, 2012). CTAT takes a much more graphical approach to constructing a tutor, whereas xPST is text-based. Both systems create tutors that are problem-specific. We had authors construct tutors in two domains, statistics and geometry. Sixteen participants were classified based on their programming background, programmer or non-programmer. Each participant was given a task of building three tutors using CTAT or xPST for a specific domain. We controlled problem complexity by ensuring that all problems could be solved using six subgoals. The instructions provided to the authors included the problem solutions and feedback to give students under certain circumstances, and also included the instructional material described above in the previous study. This approach minimized time spent by participants on pedagogical design. We evaluated the correctness of the models as well as the time spent creating each tutor. Examining how well the finished tutors instructed learners (using the metric presented in Table 2), no differences were observed between the tutors produced by the programmer and non-programmers, with all but 2 tutors (one xPST tutor and one CTAT tutor, both done by non-programmers) being judged at the highest caliber. Both tools allowed non-programmers to create effective instruction. Given the small number of participants per condition and by providing the feedback to give, making judgments concerning time is speculative. However, the CTAT authors spent less time constructing their tutors in the beginning, but by the third tutor problem there were no significant time differences, with all but one participant spending no more than 25 min per problem, regardless of tool. One xPST participant spent 133 min on the first statistics problem, hence the large average for the first problem in that condition, and another xPST participant spent 40 min on the third geometry problem. Figure 8 shows these results (the data from one non-native English speaker, who spent over 120 min on all problems, was dropped from this analysis).

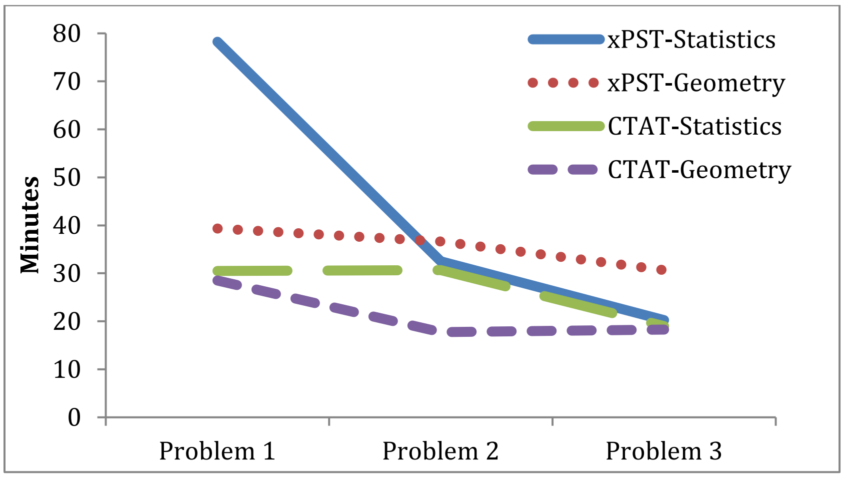


Figure 8. Average time versus problem number.

More importantly for this analysis, we chose a subset of elements from Green and Petre's cognitive dimensions framework (1996) to compare the visual programming language of CTAT with the text-based language xPST. *Closeness of mapping* is the degree to which elements and operations within the programming language can be mapped directly onto objects and actions within the problem domain. CTAT has a closer mapping than xPST since textual languages have syntax constraints that do not relate to the problem domain. *Diffuseness/terseness* relates to the number of symbols required to express a concept. Visual programming languages might suffer once the program reaches a reasonable degree of complexity. CTAT would be more diffuse than xPST when more branches or paths to a goal are included. We used relatively simple problems for this comparison. *Premature commitment* refers to the extent to which the authoring tool forces the user to make decisions before complete information is available. It often happens when components have significant interdependencies and there are no order constraints. In xPST, programmers can easily insert stubs to address this issue and return to complete later. *Secondary notation* refers to the tool's ability to convey meaning about the structure of the program with methods beyond the elements of the program itself. Indentation, commenting and blank lines can be used to illustrate the coherent chunks of the tutor on text-based tools. Some visual programming languages like CTAT support naming of states and grouping of related states, and use different colors to distinguish correct and incorrect actions in the behavior path as well. *Error-proneness* relates to the extent to which the authoring tool induces careless mistakes from the author. The syntax of text-based languages might be more error-prone, and harder to detect, especially if the errors are not syntactic in nature. *Visibility and juxtiposibility* refer to the number of steps required to make desired information visible and the ability to see separate portions of the system at the same time. The text-based systems basically have higher visibility (since all code is present) and juxtiposibility (via multiple windows), whereas in CTAT the user has to open new dialog windows to edit information within a node (lower visibility) and it is difficult to see the contents of two nodes simultaneously (lower juxtiposibility). Across these measures, each tool has some advantages according to this framework. Table 3 encapsulates these findings. For getting up-to-speed in authoring and for smaller domains, CTAT looks more advantageous, but for slightly more advanced users and larger tutoring domains, xPST might have the advantage.

Table 3. Comparison of CTAT and xPST along Green and Petre's (1996) cognitive dimensions.

Cognitive Dimension	CTAT	xPST
Closeness of Mapping	√ Visual programming allows direct mapping	
Diffuseness/Terseness	√ Diffuse for small models	√ Diffuse for large models
Premature Commitment		√ Author can leave some information blank to return later
Secondary Notation	√ Includes shapes, colors, etc.	√ Includes indentation, comments
Error Proneness	√ Not possible to have syntactic errors	
Visibility and Juxtiposibility		√ All visible in one window; juxtiposible in multiple windows

CONCLUSIONS

In developing xPST and creating the specific implementations we achieved several accomplishments, and also uncovered a number of challenges regarding the construction of such ITSs. We discuss these below.

Creating an ITS with model-tracing like capabilities can be done by non-cognitive scientists and non-programmers. While other researchers have examined this issue, including ourselves with other authoring systems, such authors using xPST were able to quickly create effective tutors. Historical estimates for creating model-tracing ITSs indicated that it took 100-300 hours of development time for each hour of instruction (Murray, 1999; Murray, 2003). While quantities of scale can mitigate this huge development cost, this large ratio would put the development of such tutors out of reach of many types of people and make it impractical for many different situations. Aleven and colleagues (2009) list the ratios for various tutors developed in CTAT. The lowest ratio in their table was 48:1. As they do, we caution against making any direct comparisons due to a variety of reasons (e.g., equating knowledge level of users, scale of development, and quality of tutor). We would characterize the ratios that we achieved in our authoring studies as consistent with those of CTAT, making the creation of such tutors much more practical by non-programmers and non-cognitive scientists in a wide variety of instances.

ITSs with limited content can be effective. Some existing tutors have many hours of content, perhaps as long as a full year's worth of high school math. Most authors do not have the time to invest, at least not initially, to produce that much material. However, an assignment that might take an hour or so for a student to complete would be within the possibility for an instructor to create. We have demonstrated that authors can quickly come up to speed in learning xPST and create sufficient instruction for it to be of use to students in learning the material.

Providing tutoring layered on existing third-party software can be done, though challenging. We have discussed this issue above in the section on the xPST architecture. However, tutoring on existing software can be powerful pedagogically, as it lowers or eliminate transfer issues for learners as they switched from being tutored on a task to being untutored (or, perhaps there would be no need to switch). However, tutoring on an interface where the full range of user interaction is possible opens up issues that usually aren't encountered in custom software. This calls for a number of different ways that the tutor may interact with the student.

FURTHER WORK

The research on xPST has also yielded several findings that point to future research directions. We discuss three of these findings below.

Task sequence affects complexity of the tutor. The xPST structure has allowed us to explore the issue of whether task sequence matters. In some tasks, sequence does matter; one cannot drive a car without finding the keys first. Sometimes it does not matter, though. At the grocery store one can get the items on the list in any order. However, task sequence might matter at the grocery store if the goal is to be most efficient. When should tutors provide feedback based on the sequence in which the learner performs actions? We think the behavior of the tutoring system depends mostly on the learning goals, and that it is not always necessary for the tutor to focus on sequence. xPST can handle this within its Sequence section, by the use of the Boolean operators (e.g., an author could say Sequence A needs to be completed OR Sequence B THEN Sequence C). By using these options within the Sequence, an author can create a tutor in which feedback and hints vary depending on the sequence of steps taken by the student. However, considering and handling multiple sequences can be challenging for authors, and model-tracing tutors need to know each possible sequence in order to trace the student's actions. A constraint-based tutor, alternatively, would have a much easier time handling such multiple paths, as these tutors are checking to make sure certain waypoints are reached within the constraints, regardless of actual path to reach them. A future research effort could focus on establishing a framework for guiding the decision of when a tutor might benefit from additional sequences, vs. a state-based or constraint-based approach.

Procedural tutors are not always the right approach. In a related point, when tasks have more than one correct answer, especially when sequence is not particularly important, a model-tracing tutor becomes less of a good fit. Model-tracing tutors typically assume that the learning task is procedural, and though the model-tracing architecture can support multiple alternative paths, paths are still the unit of analysis above individual subgoals. xPST is also designed to focus on procedures (see its reliance on the

Sequence of goalnodes). While exploring game-based tutors, ConceptGrid, and diagram tutors, it became apparent that rather than procedural paths, systems that conceptualize the learner as changing states within an unordered state-space can be more appropriate in some instances. In a serious game, for example, the tutor may not care how learners reached a certain state, but only that it was reached. Sottilare and Gilbert (2011) and Devasani, et al. (2011) describe other considerations of tutoring within games. In ConceptGrid, in which the tutor evaluates text, there is no conception of a procedure or history of previous texts, so a state space makes sense. In diagram tutors, such as Andes (VanLehn, et al., 2002), CogSketch (Forbus, Usher, Lovett, Lockwood, & Wetzel, 2011), SketchMiner (Smith, Wiebe, Mott, & Lester, 2014), and those described in Guo et al. (2014) and Guo and Gilbert (2014), the feedback is typically organized based on components of the diagram rather than time or sequence. The constraint-based tutors mentioned above (e.g., ASPIRE) also follow this state space approach. Future research might focus on designating the best tutor architecture given a particular learning domain and types of learning objectives. Or, it could be that there are possibilities to blend procedural and state-based tutor architectures.

Distinguishing exploration from incorrect paths can be challenging. This challenge is related to tutoring on third-party software. Learners should be given enough freedom to explore the problem space with little disruption from the tutor. For example, users of a new interface sometime familiarize themselves with the interface by looking through menu options or settings. Or, in a serious game environment, a learner may explore the virtual environment briefly before setting out for a goal. This exploration behavior should be allowed to an extent, but distinguishing it from off-path incorrect behavior can be difficult. Thus, for actions that don't affect the outcome of task goals, the tutor should not intervene. Instead, the tutor should take interest in only actions that are subgoal-related, e.g., actions that are not undo-able that affect the potential for desired goalnodes to exist or that prevent the learner from reaching subsequent goalnodes. xPST can handle such ambiguity to some extent, either by use of the Sequence section as mentioned above, or perhaps the use of an expert feature of the Mappings section we added (mappings are almost always a simple list of tuples, pairing interface elements with goalnodes, but it is possible to map multiple interface elements to the same goalnode via a "switch" statement). We also added a "block" flag that could be put on interface element mappings for some third-party software so that if the tutor author thought that a learner's taking a particular action would be difficult to undo, xPST could actually block the third-party software from receiving that action. For example, if a student clicked a button that would effectively erase significant previous progress, xPST could block the button click and display an error message. This approach was rarely used, as it borders on changing the interaction design of the original software, but it is a way to provide scaffolding for beginners to prevent loss of work. Future research might provide templates or general guidance to instrumenting third-party software for tutoring.

ACKNOWLEDGEMENTS

This material is based upon work supported in part by the National Science Foundation under Grants No. EEC-9876363 and IIP-0548754 and by the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or AFOSR. We thank Steven Ourada as the senior software architect of xPST.

REFERENCES

Aleven, V., McLaren, B.M., Sewall, J., Koedinger, K.R. (2009). A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education*, *19*, 105-154.

Amalathas, S., Mitrovic, A., Ravan, S. Decision-Making Tutor: Providing on-the-job training for oil palm plantation managers. *Research and Practice in Technology-Enhanced Learning, 7*(3), 131-152, APSCE, 2012.

Anderson, J. R. (1993). *Rules of the Mind.* Hillsdale, NJ: Erlbaum.

Anderson, J. R. & Pelletier, R. (1991). A development system for model-tracing tutors. In *Proceedings of the International Conference of the Learning Sciences*, 1-8. Evanston, IL.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science, 13*, 467-506.

Anderson, J. R., Corbett, A. T., Koedinger, K., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of Learning Sciences*, 4, 167-207.

Bell, P., Lewenstein, B., Shouse, A. W., & Feder, M. A. (2009). *Learning science in informal environments: People, places, and pursuits*: National Academies Press.

Blessing, S. B. (2003). A programming by demonstration authoring tool for model-tracing tutors. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp. 93-119). Netherlands: Kluwer Academic Publishers.

Blessing, S. B., Devasani, S., & Gilbert, S. (2011). Evaluation of WebxPST: A browser-based authoring tool for problem-specific tutors. In G. Biswas, S. Bull & J. Kay (Eds.), *Proceedings of the Fifteenth International Artificial Intelligence in Education Conference* (pp. 423-425), Auckland, NZ. Berlin, Germany: Springer.

Blessing, S. B., Devasani, S., & Gilbert, S. (2012). Evaluation of ConceptGrid: An authoring system for natural language responses. In *Proceedings of the Twenty-fifth International FLAIRS Conference* (pp. 426-431), Marco Island, FL. AAAI Press.

Blessing, S., & Gilbert, S. B. (2008). Evaluating an Authoring Tool for Model-Tracing Intelligent Tutoring Systems. In *Intelligent Tutoring Systems* (pp. 204-215). Berlin, Germany: Springer.

Blessing, S. B., Gilbert, S. B., Blankenship, L. A., & Sanghvi, B. (2009). From SDK to xPST: A new way to overlay a tutor on existing software. In *Proceedings of the Twenty-second International FLAIRS Conference* (pp. 466-467), Sanibel Island, FL: AAAI Press.

Blessing, S. B., Gilbert, S., Ourada, S., & Ritter, S. (2009). Authoring model-tracing cognitive tutors. *The International Journal for Artificial Intelligence in Education*, *19*, 189-210.

Brooke, J. (1996). A "quick and dirty" usability scale. In P.W. Jordan, B. Thomas, B.A. Weerdmeester & A . L. McClelland (eds.). *Usability Evaluation in Industry*. London: Taylor and Francis.

Cheikes, B. A., Geier, M., Hyland, R., Linton, F., Rodi, L., & Schaefer, H.-P. (1999). Embedded training for complex information systems. *International Journal for Artificial Intelligence in Education, 10*, 314-334.

Devasani, S., Aist, G., Blessing, S. B., & Gilbert, S. (2011). Lattice-based approach to building templates for natural language understanding in intelligent tutoring systems. In G. Biswas, S. Bull & J. Kay (Eds.), *Artificial Intelligence in Education* (pp. 47-54). Berlin, Germany: Springer.

Devasani, S., Gilbert, S., & Blessing, S. B. (2012). Evaluation of two intelligent tutoring system authoring tool paradigms: Graphical user interface-based and text-based. *Proceedings of the 21st Conference on Behavior Representation in Modeling and Simulation* (pp. 54-61), Amelia Island, FL.

Devasani, S., Gilbert, S., Shetty, S., Ramaswami, N., & Blessing, S. B. (2011). Authoring intelligent tutoring systems for 3D game environments. Paper presented at the First Workshop on Authoring Simulation and Game-based Intelligent Tutoring at the Fifteenth International Artificial Intelligence in Education Conference. Auckland, NZ.

Forbus, K., Usher, J., Lovett, A., Lockwood, K., & Wetzel, J. (2011). CogSketch: Sketch Understanding for Cognitive Science Research and for Education. *Topics in Cognitive Science, 3*(4), 648-666.

Gee, J. P. (2005). Learning by design: Good video games as learning machines. *E-learning, 2*(1), 5-16.

Gilbert, S. B., Blessing, S. B., & Blankenship, E. (2009). The accidental tutor: Overlaying an intelligent tutor on an existing user interface. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems* (pp. 4603-4608). New York: ACM.

Gilbert, S., Blessing, S. B., & Kodavali, S. (2009). The Extensible Problem-specific tutor (xPST): Evaluation of an API for tutoring on existing interfaces. In V. Dimitrova et al. (Eds.), *Proceedings of the 14th International Conference on Artificial Intelligence in Education* (pp. 707-709), Brighton, UK. Amsterdam, Netherlands: IOS Press.

Gilbert, S., Devasani, S., Kodavali, S., & Blessing, S. B. (2011). Easy authoring of intelligent tutoring systems for synthetic environments. *Proceedings of the 20th Conference on Behavior Representation in Modeling and Simulation* (pp. 192-199), Sundance, UT.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing, 7*, 131- 174.

Guo, E., & Gilbert, S. (2014). *Instructional Strategies in Diagram-based ITS: Lessons Learned from Two Tutoring Systems* Paper presented at the International Conference on Intelligent Tutoring Systems: Workshop on Pedagogy that Makes a Difference: Exploring Domain-Independent Principles across Instructional Management Research within the ITS Community, Honolulu, HI.

Guo, E., Gilbert, S., Jackman, J., Starns, G., Hagge, M., Faidley, L., & Amin-Naseri, M. (2014). *StaticsTutor: Free Body Diagram Tutor for Problem Framing.* In *Intelligent Tutoring Systems* (pp. 448-455). Berlin, Germany: Springer.

Hategekimana, C., Gilbert, S. & Blessing, S. (2008). Effectiveness of using an intelligent tutoring system to train users on off-the-shelf software. In K. McFerrin et al. (Eds.), *Proceedings of Society for Information Technology and Teacher Education International Conference 2008* (pp. 414-419), Las Vegas, NV. Chesapeake, VA: AACE.

Hayashi, Y., Bourdeau, J., & Mizoguchi, R. (2009). Using ontological engineering to organize learning/instructional theories and build a theory-aware authoring system. *International Journal of Artificial Intelligence in Education*, *19*(2), 211-252.

Johnson, B.G., & Holder, D. A. (2010). A model-tracing intelligent tutoring system for assigning oxidation numbers in chemical formulas. *The Chemical Educator*, *15*, 447-454.

Johnson, W. L., Rickel, J. W., & Lester, J. C. (2000). Animated pedagogical agents: Face-to-face interaction in interactive learning environments. *International Journal of Artificial Intelligence in Education*, *11*, 47-78.

Johnson, W. L., & Valente, A. (2009). Tactical language and culture training systems: Using AI to teach foreign languages and cultures. *AI Magazine, 30*(2), 72.

Kang, H., Plaisant, C., & Shneiderman, B. (2003). New approaches to help users get started with visual interfaces: multi-layered interfaces and integrated initial guidance. In *Proceedings of the 2003 Annual National Conference on Digital Government Research* (pp. 1-6). Digital Government Society of North America.

Kim, J. M., Hill, R. W., Durlach, P. J., Lane, H. C., Forbell, E., Core, M., Marsella, S., Pynadath, D., & Hart, J. (2009). BiLAT: A game-based environment for practicing negotiation in a cultural context. *International Journal of Artificial Intelligence in Education*, *19*, 289-308.

Kodavali, S., Gilbert, S., Blessing, S. B. (2010) Expansion of the xPST framework to enable non-programmers to create intelligent tutoring systems in 3D game environments. In V. Aleven, J. Kay, & J. Mostow (Eds.), *Proceedings of the 10th International Conference on Intelligent Tutoring Systems* (pp. 365-367), Pittsburgh, PA. Berlin, Germany: Springer.

Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1997). Intelligent tutoring goes to school in the big city. *International Journal of Artificial Intelligence in Education, 8*, 30-43.

Kumaraguru, P., Rhee, Y., Acquisti, A., Cranor, L. F., Hong, J., & Nunge, E. (2007). Protecting people from phishing: the design and evaluation of an embedded training email system. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 905-914). NY: ACM.

Leelawong, K., & Biswas, G. (2008). Designing learning by teaching agents: The Betty's Brain system. *International Journal of Artificial Intelligence in Education*, *18*, 181-208.

Livak, T., Heffernan, N. T., & Moyer, D. (2004). Using cognitive models for computer generated forces and human tutoring. In *13th Annual Conference on (BRIMS) Behavior Representation in Modeling and Simulation. Simulation Interoperability Standards Organization.* Arlington, VA. Summer 2004.

Maass, J. K., & Blessing, S. B. (2011). xSTAT: An intelligent homework helper for students. Poster presented at the 2011 Georgia Undergraduate Research in Psychology Conference, Kennesaw, GA.

Martin, B., & Mitrovic, A. (2002). WETAS: A web-based authoring system for constraint-based ITS. In *Adaptive Hypermedia and Adaptive Web-Based Systems* (pp. 543-546). Berlin, Germany: Springer.

Martin, B., Mitrovic, A., & Suraweera, P. (2007). *Domain modelling with ontology: A case study*. Presented at A3EH, 5th Adaptive Authoring for Educational Hypermedia, Workshop AH.

Mitrovic, A., Martin, B., Suraweera, P., Zakharov, K., Milik, N., Holland, J., & McGuigan, N. (2009). ASPIRE: an authoring system and deployment environment for constraint-based tutors. *International Journal of Artificial Intelligence in Education*, *19*(2), 155-188.

Munro, A. (2003). Authoring simulation-centered learning environments with RIDES and VIVIDS. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp. 61-91). Netherlands: Kluwer Academic Publishers.

Murray, T. (1999). Authoring Intelligent Tutoring Systems: An analysis of the state of the art. *International Journal of AI in Education, 10*, 98-129.

Murray, T. (2003). An overview of intelligent tutoring system authoring tools: Updated analysis of the state of the art. In T. Murray, S. Blessing & S. Ainsworth (Eds.) *Authoring Tools for Advanced Learning Environments* (pp. 491-544). Dordrecht, the Netherlands: Kluwer Academic Publishers.

Ritter, S., & Blessing, S. B., Wheeler, L. (2003). User modeling and problem-space representation in the tutor runtime engine. In P. Brusilovsky, A. T. Corbett , & F. de Rosis (Eds.), *User Modeling 2003* (pp. 333-336). Berlin, Germany: Springer.

Ritter, S., & Koedinger, K. R. (1996). An architecture for plug-in tutor agents. *Journal of Artificial Intelligence in Education, 7*(3), 315-347.

Ritter, S., Kulikowich, J., Lei, P., McGuire, C.L. & Morgan, P. (2007). What evidence matters? A randomized field trial of Cognitive Tutor Algebra I. In T. Hirashima, U. Hoppe & S. S. Young (Eds.), *Supporting Learning Flow through Integrative Technologies* (Vol. 162, pp. 13-20). Amsterdam: IOS Press.

Roselli, R.J., Gilbert, S., Howard, L., Blessing, S. B., Raut, A., & Pandian, P. (2008). Integration of an intelligent tutoring system with a web-based authoring system to develop online homework assignments with formative feedback. In *Proceedings of 115th Annual American Society for Engineering Education Conference (ASEE 2008).*

Shute, V. J., Ventura, M., Bauer, M., & Zapata-Rivera, D. (2009). Melding the power of serious games and embedded assessment to monitor and foster learning. *Serious games: Mechanisms and effects*, 295-321.

Smith, A., Wiebe, E., Mott, B., & Lester, J. (2014). *SKETCHMINER: Mining Learner-Generated Science Drawings with Topological Abstraction.* Paper presented at the The Seventh International Conference on Educational Data Mining, London, England.

Sottilare, R. and Gilbert, S. B. (2011). Considerations for adaptive tutoring within serious games: authoring cognitive models and game interfaces. Presentation at the Authoring Simulation and Game-based Intelligent Tutoring Workshop at the Fifteenth Conference on Artificial Intelligence in Education, Auckland, NZ.

Towne, D. M. (2003). Automated knowledge acquisition for intelligent support of diagnostic reasoning. In T. Murray, S. Blessing, & S. Ainsworth (Eds.) *Authoring Tools for Advanced Technology Learning Environments* (pp. 121-147). Netherlands: Kluwer Academic Publishers.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005). The Andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence and Education, 15*(*3*), 147-204.

VanLehn, K., Lynch, C., Taylor, L., Weinstein, A., Shelby, R., Schulze, K., et al. (2002). Minimally Invasive Tutoring of Complex Physics Problem Solving. In S. Cerri, G. Gouardères & F. Paraguaçu (Eds.), *Intelligent Tutoring Systems* (Vol. 2363, pp. 367-376). Berlin, Germany: Springer.