Plug-in Tutor Agents: Still Pluggin’

Steven Ritter¹

Published online: 10 September 2015

© International Artificial Intelligence in Education Society 2015

Abstract *An Architecture for Plug-in Tutor Agents* (Ritter and Koedinger 1996) proposed a software architecture designed around the idea that tutors could be built as plug-ins for existing software applications. Looking back on the paper now, we can see that certain assumptions about the future of software architecture did not come to be, making the particular approach described in the paper infeasible. However, the pedagogical approach assumed by the architecture remains relevant today, and the basics of the architecture are applicable in purpose-built instructional systems.

Keywords Intelligent tutoring systems · Authoring tools · Software architecture

It is an honor for *An Architecture for Plug-in Tutor Agents* to be included in this special issue. In re-reading the paper, I was struck by the vision expressed in the paper, which extended to both general software architecture and to a pedagogical approach to tutoring. The paper’s strength is in articulating how a particular approach to instruction can be translated to a practical and fairly general software architecture. The pedagogical approach drew from a long history of cognitive apprenticeship or, more generally, the idea that instruction is most effective when it takes place in a context that is maximally similar to the intended context of use. If this transfer context involved the use of software, then the natural approach is to embed the instructional content directly within the existing software.

The paper conceptualizes this process as “plugging in” a tutoring component to pre-existing software, using Microsoft Excel and Geometer’s Sketchpad as examples. The paper discusses many practical and theoretical considerations involved with making such a plug-in work, ending with a detailed description of a general software architecture that articulated the particular roles of the “tool” (pre-existing software) and tutor (the instructional component) and the way that they interact.

✉ Steven Ritter
sritter@carnegielearning.com

¹ Carnegie Learning, Pittsburgh, PA, USA

The resulting component-based software solution envisioned assembling pre-built modules using standard modes of interaction and embedding. A more far-reaching and, to this date, unsuccessful extension of this approach was to develop standard software mechanisms for these modules to control and, more importantly for the tool-tutor architecture, monitor each other. While this software vision did not come to pass in the way many envisioned at the time, the component-based architecture is still the basis for current tutoring systems, even when they do not incorporate off-the-shelf software.

With respect to pedagogy, the vision in the tool-tutor architecture paper is still very much with us. The paper didn't solve all issues related to providing instruction in the context of using real-world tools, but it did introduce and provide some direction for thinking about how to approach these problems.

Context

Three developments are important for understanding the basis for this paper: the state of Cognitive Tutors, new approaches to pedagogy and developments in software architecture.

Cognitive Tutors

At the time this paper was written, Cognitive Tutors had been under development for about 10 years (Anderson 1983), and they had started to achieve some success in regular use within classrooms (Koedinger et al. 1997). This initial success led us to think about how to scale up this success, both in terms of distributing and supporting the curricula (which is what led to Carnegie Learning) and in terms of the technology. This transition led to a summing-up paper (Anderson et al. 1995), which presented eight principles for tutor design. The first principle was "Represent student competence as a production set." The idea of representing competence as productions followed from the basis of the tutors in Anderson's ACT-R cognitive architecture.

Production systems consist of stored knowledge and procedures that act on new information and stored knowledge in order to make decisions. Within Cognitive Tutors, knowledge was modeled with the Tutor Development Kit (TDK; Anderson and Pelletier 1991). The experience of writing a production system requires the author to clearly delineate exactly what elements of the problem are essential for the student to encode, what elements of knowledge the student needs to have in order to successfully solve the problem and what processes or procedures the student must use to progress from the initial problem state to the final state. The process of representing competence as a production system requires the author to think deeply about the way that students solve problems, and this process forms the basis of the recommendation.

The TDK implemented this analysis of competence in a rule-based system. While this is a very powerful approach, it is a specialized kind of programming, and it can be computationally expensive. A rule-based system considers what actions should be taken, given the current state of the system. The current state consists of both elements of the problem representation and elements of student knowledge (including the work that a student has already done on the problem). For example, in an equation like " $X + 4 = 5$ ", the system would ask itself problem information such as whether the equation

was linear, whether the constant was positive, etc. in addition to student knowledge such as whether the student was able to represent the goal of isolating X on one side of the equation. As the domains considered by the tutors broadened, this decision-making became more computationally expensive, so we started to think about software architectures that would preserve the pedagogical benefits of model-tracing tutors without the computational cost.

Our solution drew inspiration from Herb Simon's story about an ant on the beach (Simon 1996). Simon posed the problem of trying to predict the trajectory of an ant walking across a beach. While the trajectory is partially determined by ant psychology, most of the trajectory is the ant responding rationally to the constraints of the environment: the particular small hills and valleys formed by the grains of sand. An analysis of the problem-solving done within Cognitive Tutors similarly led to the conclusion that most of the reasoning done by the production system (and, by implication, the student) had to do with aspects of the problem posed to students, not aspects of student thinking. This insight led to the development of the Tutor Runtime Environment (TRE; Ritter et al. 2003), which, essentially, compiles out the static problem information from the production system, leaving a problem-specific runtime system that only needs to consider the problem-solving elements that depend on student knowledge. As in the TDK, the TRE needs to consider both elements of the problem and elements of the student's knowledge. The distinction is *when* these aspects are considered. The TRE determines at compile-time that " $X+4=5$ " involves a positive constant, and it produces a runtime representation that only operates on that equation (as well as similar runtime systems that operate on other equations). This greatly simplifies the runtime processing.

The need to solve a performance issue led to a reconsideration of the software architecture underlying the tutors. Within the TDK, there was a tight connection between the production system and user interface elements. We wanted to re-use our existing interfaces within the TDK but also transition the tutoring logic to the TRE. This required us to think about the tutor as a software component, independent from the interface, and to think about the API that would structure communication between the interface and the tutor. This software need naturally led to thinking about the tutoring component as an element separate from the user interface.

Pedagogy

Another principle for cognitive tutoring provided in Anderson et al. (1995) was "Provide instruction in the problem-solving context." This principle followed from approaches like cognitive apprenticeship (Collins et al. 1989) but also from research demonstrating difficulty in transferring knowledge from one situation to another (Anderson 1990). The best learning environment is one that is maximally similar to the transfer environment. As computer-based tools (like Microsoft Excel) increasingly became the transfer environment, it made sense to consider using those tools as part of the instructional system.

We developed plug-in tutors for two kinds of software tools (Microsoft Excel and Geometer's Sketchpad) in part to test the adequacy of the architecture from a technical perspective but also because these tools represented two different pedagogical approaches to using off-the-shelf software. Geometer's Sketchpad was developed as an educational tool. The role of the plug-in agent was to transform an unguided,

discovery-based environment into one that was more scaffolded. Our use of Microsoft Excel was very different. Our tutor agent was not focused on teaching Excel itself (though later work did take this approach, c.f., Mathan and Koedinger 2005); instead, we developed a tutor to teach students how to solve linear word problems, using Microsoft Excel as an interface. The flexibility of Excel lends itself to this use and Excel is a real-world problem-solving context that is used to reason about linear relationships, but Excel is not designed solely for this use. The scope and focus of the tool is so broad that a major design challenge was to constrain off-task behavior so that students would not wander so far off track that they could not find their way back to a reasonable solution.

Software Architecture

The belief that it was feasible to use existing software tools as part of an instructional system was supported by an emerging vision of software development that included standards for linking and embedding independently developed software (the OpenDoc and OLE systems) and for controlling software through scripting systems.

The idea behind component systems like OpenDoc and OLE was that developers could produce functional tools (like a graphing system or a spreadsheet) that could be embedded in other documents (such as a word processing document) with full functionality. The vision was that the end user could produce an office suite or other complex software application by embedding components that fit their needs. The idea that this could be done in a standard way was an inspiration for the idea that a tutoring component could be added to an existing system. Perhaps, even, users could choose from a set of different tutoring systems for the same component, choosing one that best fit their own educational goals and needs.

The cognitive apprenticeship pedagogy supported a model where the tutor would act like an expert looking over the shoulder of a student, interrupting, commenting on or correcting the student's work only when necessary. But how could a software component "observe" the work that a student was doing in an existing software tool like Microsoft Excel? One significant complication in such observation is that any particular domain goal can typically be achieved in many different ways. For example, the goal of summing numbers in a column in an Excel spreadsheet could be achieved by selecting the cells and hitting the "sum" icon in the toolbar, by selecting the cell below the numbers to be added and hitting the "sum" icon (which intelligently figures out the range of cells to be summed), by typing in a formula and several other ways. Each of these methods, in turn, has many variations at the interface level (such as typing or selecting the range of cells to be summed in the formula).

Our goal was to use Microsoft Excel as a tool for doing mathematics, and so we were not concerned with the particular way that the user chose to sum a column of numbers. Instead, we were concerned that the user recognized that, in order to solve a particular math problem, they needed to sum the column of numbers. Our tutoring system would need to be enormously more complex, if it needed to understand everything about how Excel's user interface could be used to sum a column of numbers. What we really wanted was a way for Excel to tell us the semantics of a user action (summing a column of numbers), rather than the keystrokes and mouse actions that were used to achieve that action.

Fortunately, our desire coincided with another vision of the future of software: the ability to control software through a general scripting knowledge. Our work followed Apple's approach with AppleScript (which was itself based on Hypercard scripting), but a similar vision existed on the Microsoft side and was partially implemented as Visual Basic for Applications, a scripting system that was, in theory, available to all but, in practice, was mostly limited to the Microsoft Office suite. I'm not sure how many readers of the initial paper realized that the reference to Gates (1987) was, in fact, Bill Gates writing in *Byte Magazine* about his vision of a scripting system that could link applications. In both Apple and Microsoft's vision, the scripting system involved controlling the semantics of the application (not just the user interface) and included that idea that, if applications could respond to semantic commands, they could just as easily emit such commands, which could be "observed" by another application. The observing application was envisioned to be a script-writing system, so that users could demonstrate some action and use the record of the events used as the basis for a script, but we thought that we could just as well make the tutoring system an observing application. In this way, we could leverage this observational capability of the operation system to implement a tutoring system that could watch students use an existing application without the application requiring any code or functionality focused directly on supporting a tutor. The fact that the scripting system could also affect the state of the application gave us a way to pose problems and present feedback to students.

Both component systems and scripting systems proved too difficult to implement for application software developers, and the demand for them was too limited to support them in the form envisioned in the 1990s. More limited versions of systems like these still exist (for example, plug-in keyboards on Android and IOS are examples of this type of component embedding; IOS also supports UI automation scripting through JavaScript and OS X's Automator has a recording facility at the user-interface level) but the vision that such systems would be fully implemented and standard across wide ranges of applications never came to pass. Instead, such systems tend to be used in limited cases (such as in page layout systems) or to support features like accessibility.

The Approach

In re-reading it, I have the sense that, although the paper directly addresses tutoring of off-the-shelf software, our real target was somewhat broader. In fact, although the paper addresses the possibility of adding tutoring agents to off-the-shelf software (through technologies like AppleEvent Recording that didn't quite play out the way we'd hoped), the broader target and the one that has had more lasting influence is in building tutor agents for tutoring environments that, while not off-the-shelf, were to be built as if they were independent from the tutor agent and follow the general pedagogical approach inherent in the architecture. The idea here is to separate the development of the tool from that of the tutor agent. This allows the tool to be developed to be somewhat more powerful than is technically necessary to implement the initial version of the tutoring system. As the system gets developed, deployed and tested, the tutor agent developer(s) then have the flexibility to change the way that errors are indicated or feedback is given, without making changes in the tool.

You can see a suggestion of this approach in the two main examples: the Excel tutor and the Geometer's Sketchpad. In both cases, the tools were not really off-the-shelf. The Geometer's Sketchpad (<http://www.keycurriculum.com>) is a virtual geometry construction tool that allows students to (among other things) use a virtual compass and straightedge to create geometric constructions that are dynamic. When the student changes one geometric element elements of the construction that are mathematically determined are also changed. For example, the student could construct a set of parallel lines with a transversal and, by changing the angle at which the transversal intersects the parallel lines, observe that alternate interior angles remain equal. Geometer's Sketchpad needed to be modified in order to add a "tutor" menu that allowed the student to directly ask the tutor agent for a hint. We also struggled with a way to present textual hints within Sketchpad. We could arrange for a messages window belonging to the tutor application (not Sketchpad) to be visible when we had a message to display, but the solution was awkward, due to application layering issues.

Excel was (and is) a highly programmable application, and we used programmability features to communicate user actions to the tutor (duplicating the functionality of AppleEvent Recording, which was not implemented in Excel at the time) and to add a tutor menu to the interface.

So, even in the prime examples illustrating tutor agents for off-the-shelf software, the off-the-shelf software needed to be modified in order to support the tutor. There was some hope that OpenDoc (or a similar system) would help resolve some of these kinds of issues, by allowing us, for example, to create a "wrapper" application that contained both Sketchpad and a messaging interface into a container application, much the way that a web page can be "framed" in a larger context.¹ This would require tool-side authoring but would at least reuse existing tools. In retrospect (and, perhaps, at the time) the more realistic target for the architecture was a tutoring environment built from scratch but architected with the belief that the interface should be developed separate from the tutor and that the two should be connected through a clear messaging protocol.

Considered from this perspective, four aspects of the tool-tutor approach have been especially influential on current approaches to tutoring.

Tools and Interfaces

The first aspect of the tool-tutor environment is that the tool is not just an interface. The tool should be considered as an application that supports real work, independent of the tutoring context. As such, the tool has inherent computational capabilities which are distinct from any educational context. When the value of a cell changes, Excel (or a tool acting like a spreadsheet) should be responsible for calculating dependent cells. The specifics of this calculation (such as maintaining dependent relationships among cells) are the natural domain of the tool, regardless of whether the tool is being used in an educational context. Since the computational capabilities of many tools are extensive, re-implementing them in an educational system would be very expensive. The plug-in architecture provides a model in which there is no need for the tutor to maintain a complete model of the effects of a student action in the tool.

¹ Gilbert et al. (2009) have pursued this approach to tutoring existing web-based material.

The tutor, on the other hand, is responsible for understanding the student's aim in using the tool. If the tutor knows what the student is trying to accomplish, it can determine whether the particular steps that the student takes are on the path to that goal.

Our original paper refers to several help systems (including EuroHelp and AppleGuide²), which were inspirations for the approach. The references to help systems also served as a contrast to goal-based tutoring systems. Help systems can be considered to be environments that provide assistance on using the tool. To these systems, the goals have to do with understanding which widgets and interface actions in the tool should be used to make the tool work, but these systems do not have knowledge of why the user is using the tool in the first place. Help systems care about interface actions more than the underlying semantics.

Semantic Messaging

A key aspect of the decoupling of tool and tutor was that the communication between the two happen at the “semantic” level. Semantic-level communication allowed changes in both tool and tutor without having to modify both components together. The plug-in architecture's send-message command would present a help message to the user of the system. This command did not need to worry about the specific method that the tool used to display messages. Depending on the form of the tool, such messages could be put in a floating window or in a status bar, for example. Semantic messaging standardized the method of describing an action (as a selection-action-input triple).

The alternative to semantic messaging would be messaging at the user interface level. In user interface messaging, the tutor agent would, for example, specify that the window to present the message to the student should have particular aspects (perhaps including size and screen location). Our choice of semantic messaging was driven by a desire to have tool and tutor agent focus on the aspects of instruction that seemed to naturally be in their domain. The tutor agent knows when to give feedback and what that feedback should communicate to the student. The tool knows how large the screen is, what information is currently displayed on the screen and the standard user interface conventions for displaying windows. Semantic messaging allows the tutor agent to focus on the content of feedback, leaving decisions about the form of that feedback to the tool.

Leaving display details up to the tool limits the number of messages used in the system. A single “flag” message served to indicate to the tool that an element of the interface should be highlighted as an error to the student. The details about whether such flagging should involve circling the element in red, displaying in bold or some other choice was up to the tool.

Part of the power of this semantic messaging system derives from the ability to refer to tool objects in an extendable way. We used a container system to identify objects. In

² The reference to AppleGuide may be particularly obscure to current readers. AppleGuide was a very innovative system that could be used to create step-by-step guides for accomplishing various tasks. The help author could define “coachmarks” that would accompany steps. For example, AppleGuide could put a red circle around the button that the user should press at a particular step or show an arrow pointing to a menu item. Such coachmarks helped to inspire the “point-to” and “flag” verbs described in the paper. It seemed like a great idea but it flopped so badly and was so short-lived that the internet barely remembers it (beyond a short Wikipedia article).

such a system, objects are (conceptually) contained in one or more other objects and are identified within their container either by a unique name within the container or with a fixed position. For example, a cell in Excel might be identified as “Cell named R1C1 in Window 1 in Excel”, where “R1C1” is a unique name for the cell and window reference using ordering (with “Window 1” indicating the frontmost window). This method is similar to that currently used in the Document Object Model in HTML.

Step-Based Tutoring

The tool-tutor architecture was designed for step-based tutoring, in which the tutor is expected to react (or else decide not to react) whenever the student performs some action in the tool. In re-reading the paper, I was surprised that we had not mentioned simulations, since I remember considering situations at the time the paper was written. Simulations would be particularly interesting aspect of the architecture, since they are cases where the tool may change simply due to the passage of time and without any action on the part of the student. In some cases, it would be acceptable, even in a simulation, for the tutor to only “cycle” (consider student input) when the student performs some action. But training on some simulations might want the tutor to react even when the student has done nothing (for example, the tutor might inform the student that she *should* have done something at a particular point in the simulation). The architecture’s *process-tool-action* message could be extended to include events triggered by the simulation itself, rather than the student, but this was not explicitly treated in the paper.

Tutoring Patterns

A fundamental assumption of the pedagogy behind the tool-tutor architecture is that the student is free to use the tool, and the tutor acts as a (mostly) silent observer. We can describe this approach as a “commentary pattern,” where the tutor’s role is to comment on actions that the student has taken in the tool. The commentary pattern assumes that the action that the student has taken in the tool (for example, changes to a cell value, resulting in the spreadsheet calculating dependent cells) is completed before the tutor has a chance to comment. The paper spends some time discussing a consequence of this pattern: that, if the user is able to act quickly (relative to the tutor’s ability to react), the tool and tutor can get out of sync. It can be problematic if the tutor is commenting on an action that the student took three steps ago. In practice, we have found ways to “lock” the tool until the tutor reacts, solving the synchronization problem.

A more fundamental issue with the commentary pattern is that it doesn’t deal with cases where the appropriate instructional response is to prevent student action. The paper discusses a number of cases where the tutor would instruct the tool to “undo” the student’s action, in order to restore the previous state. This approach is workable but inelegant for actions that can be undone. A more serious issue is what to do about actions that cannot be undone (which always seem to be present). What should the tutor do if, for example, the student closes a window in Excel?

For this reason, we later considered a “permission pattern,” in which the tool communicates the user’s desire to carry out an action (closing a window or changing the value of a cell), but the effects of that action await approval from the tutor. This

pattern violates the principle that the tool can be designed without any consideration that it would be used in a tutoring context but, once you think of the tool-tutor architecture as describing a methodology for developing tools for use in an instructional system (rather than as a way of using off-the-shelf tools in an instructional context), implementing this pattern seems very reasonable.

Impact

The particular goals addressed in *An Architecture for Plug-in Tutor Agents* remain today. Current tutoring architectures also strive to reduce costs through modular architecture and reuse of existing components. The drive to build training around existing tools and to embed real tools in training environments is also an important current goal.

The Cognitive Model SDK (Blessing et al. 2009) is a direct descendant of the tool-tutor architecture, as further developed in the Tutor Runtime Engine (Ritter et al. 2003). A variant of this system is currently used to build Cognitive Tutors at Carnegie Learning. For reasons related to synchronization, and support of the “permission pattern,” current tutors employ a range of strategies from ones where the user interface is considered a true tool with its own state and capabilities to ones where the tutor maintains all state and the interface is largely driven by the tutor. Although purpose-built tools do imitate aspects of off-the-shelf tools like Sketchpad and Excel, for both licensing and technical reasons, Carnegie Learning has not employed off-the-shelf software as tools within its software.

One reason to maintain the distinction between tool and tutor in the Cognitive Tutor system has been to address technical migration of the software. The first versions of the software were built in the TDK (Anderson and Pelletier 1991) and were programmed entirely in Lisp. The first applications of the tool-tutor architecture supported cross-platform (Windows and MacOS) deployment by implementing tools in Java and maintaining the Lisp back-end. This was a reasonable path because the interface-less tutor agent was relatively easy to port from one platform to another and development in Java allowed common interfaces across the two target platforms. Later versions implemented the TRE tutor-agent in Java but maintained a strict separation between tool and tutor.

Similar considerations led to maintaining a separation between tool and tutoring in the Cognitive Tutor Authoring Tools (CTAT; Aleven et al. 2009). This separation has allowed the CTAT architecture to support various combinations of user-interface and tutor agent implementations (including Flash, Java, Javascript and Lisp). By using a messaging interface between tool and tutor (with semantics similar to that described in the original paper), CTAT has been able to support different deployment architectures. In some cases, tool and tutor both run on the client, with messaging between them either within- or between-processes. In others, the tutor runs on the client, with the messaging taking place over HTTP. This flexibility has allowed researchers to quickly prototype and deploy tutors in a wide variety of domains.

The goal of monitoring user actions in off-the-shelf software without modifying the software itself continues, but the mechanisms to do this have changed substantially. We leveraged event recording, which operating systems provided for the purposes of

creating cross-application scripting. Such scripting was never very widely used and so many applications did not support it. Many current systems leverage operating system hooks and accessibility interfaces to accomplish similar goals. I now believe that leveraging existing mechanisms (whether script recording or accessibility) is problematic for widespread deployment, due to inherent differences between the educational context and the explicit purpose of the mechanism. However, limited-use and research systems have very successfully made use of these kinds of mechanisms.

The AppMonitor system (Alexander et al. 2008) uses a combination of accessibility interfaces and monitoring of low-level device drivers to monitor applications. Patina (Matejka et al. 2013) uses similar inputs, as well as window position information, to understand and present a summary of user actions. Both systems are focused on usability studies, with the result that such systems are intended to analyze user behavior after the fact, which reduces the requirement of semantic interpretability. Patina does dynamically generate an overlay summarizing user actions. SEPIA (Ginon et al. 2014) uses accessibility hooks to observe application usage for instructional purposes. This system has been applied to a wide range of applications, including both tool-based help and domain tutoring, and has made impressive progress towards providing semantic interpretations of user actions that can be used to provide assistance. In addition to its use of semantic monitoring, this system follows the tool/tutor approach in other ways, such as the separation between tool and tutor, though the specific implementation is quite different.

One modern architecture that has been especially influential and innovative in addressing modular architecture, reuse and use of off-the-shelf tools is GIFT (Sottolare et al. 2012). The GIFT architecture provides mechanisms to build tutoring based on existing systems. GIFT has been more ambitious in some ways than the tool-tutor architecture by focusing both on traditional declarative content such as that presented in Microsoft Powerpoint and on immersive simulation-based systems. One big difference between the GIFT architecture and that described in the tool-tutor architecture is that GIFT does not treat the tutor agent as a black box. Instead, GIFT defines pedagogical and domain modules, which together embody domain-independent and domain-dependent tutoring strategies. This approach has the potential to further save costs by re-using pedagogical modules across systems and by incorporating multiple domain modules, perhaps specialized for particular sub-domains, but it will accomplish this goal only to the extent that domain-independent tutoring strategies are widely applicable and to the extent that authors believe that the work required to comply with the GIFT architecture results in substantial benefits.

Conclusion

In retrospect, *An Architecture for Plug-in Tutor Agents* still provides a good high-level summary of many technical and pedagogical considerations relevant to building tutoring systems. Current architectures address many of the same issues in many of the same ways. The paper expresses the belief that productivity software need not be built with instructional considerations in mind, but that the ability to observe and script such systems will allow instructional designers to plug-in instructional components to such systems. As a robust software architecture, I believe we have not reached the point

where we can reliably depend on such “hooks” to support instruction, except in specific circumstances. However, the architecture has retained its influence as a way of constructing instructional systems. Many such systems are built as a robust and powerful tool layer attached to a tutoring component that is responsible for feedback and scaffolding of student use of the tool. This architecture both results in a more flexible system and one that can appropriately divide the labor for building such systems, with some developers focused on tool and user interface development, with others focused on instructional pedagogy. Authoring tools for each of these tasks have improved greatly over the years, leading to more cost-effective methods of producing scalable intelligent tutoring systems and other advanced instructional systems.

References

- Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2009). Example-Tracing Tutors: A New Paradigm for Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 19, 105–154.
- Alexander, J., Cockburn, A., & Lobb, R. (2008). AppMonitor: a tool for recording user actions in unmodified Windows applications. *Behavior Research Methods*, 40(2), 413–421.
- Anderson, J. R. (1983). *The architecture of cognition*. Cambridge: Harvard University Press.
- Anderson, J. R. (1990). *The adaptive character of thought*. Hillsdale: Lawrence Earlbaum Associates.
- Anderson, J. R., & Pelletier, R. (1991). A development system for model–tracing tutors. In L. Birnbaum (Ed.), *Proceedings of the International Conference of the Learning Sciences* (pp. 1–8). Charlottesville: Association for the Advancement of Computing in Education.
- Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, 4(2), 167–207.
- Blessing, S., Gilbert, S., Ourada, S., & Ritter, S. (2009). Authoring model-tracing cognitive tutors. *The International Journal for Artificial Intelligence and Education*, 19, 189–210.
- Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the crafts of reading, writing and mathematics. In L. B. Resnick (Ed.), *Knowing, learning and instruction: Essays in honor of Robert Glaser* (pp. 453–494). Hillsdale: Lawrence Earlbaum Associates.
- Gates, B. (1987). Beyond macro processing. *Byte*, 12(7), 11–16.
- Gilbert, S., Blessing, S. B., *Kodavali, S. (2009). The Extensible Problem-Specific Tutor (xPST): Evaluation of an API for Tutoring on Existing Interfaces. *Proceedings of the 14th International Conference on Artificial Intelligence in Education*. (pp. 707–709). Amsterdam, Netherlands: IOS Press.
- Ginon, B., Thai, L. V., Jean-Daubias, S., Lefevre, M & Champin, P-A. (2014). Adding epiphytic assistance systems in learning applications using the SEPIA system. In *9th European Conference on Technology Enhanced Learning, EC-TEL 2014, Graz, Austria*, pp. 138–151.
- 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.
- Matejka, J., Grossman, T. & Fitzmaurice, G. Patina: Dynamic Heatmaps for Visualizing Application Usage. *CHI 2013 Conference Proceedings: ACM Conference on Human Factors in Computing Systems*, pp. 3227–3236.
- Mathan, S. A., & Koedinger, K. R. (2005). Fostering the intelligent novice: Learning from errors with metacognitive tutoring. *Educational Psychologist*, 40(4), 257–265.
- Ritter, S., & Koedinger, K. R. (1996). An architecture for plug-in tutor agents. *Journal of Artificial Intelligence in Education*, 7, 315–347.
- 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 Rosi (Eds.), *User Modeling 2003* (pp. 333–336). Johnstown: Berlin: Springer-Verlag.
- Simon, H. A. (1996). *The sciences of the artificial*. Cambridge: MIT Press.
- Sottolare, R. A., Brawner, K. W., Goldberg, B. S. and Holden, H. K. (2012). The Generalized Intelligent Framework for Tutoring (GIFT). Orlando, FL: U.S. Army Research Laboratory – Human Research & Engineering Directorate (ARL-HRED).