Integrating GIFT and AutoTutor with Sharable Knowledge Objects (SKO)

Benjamin D. Nye

Institute for Intelligent Systems
University of Memphis, Memphis, TN 38111
benjamin.nye@gmail.com

Abstract. AutoTutor and the Generalized Intelligent Framework for Tutoring (GIFT) are two separate projects that have independently recognized the need for greater interoperability and modularity between intelligent tutoring systems. To this end, both projects are moving toward service-oriented delivery of tutoring. A project is currently underway to integrate AutoTutor and GIFT. This paper describes the Sharable Knowledge Object (SKO) framework, a service-oriented, publish and subscribe architecture for natural language tutoring. First, the rationale for breaking an established tutoring system into separate services is discussed. Secondly, a short history of AutoTutor's software design is reviewed. Next, the design principles of the new SKO framework for tutoring are described. Finally, the plans and progress for integration with the GIFT architecture are presented.

Keywords: Intelligent Tutoring Systems, Service Oriented Architectures, Message Passing, Design Patterns, Systems Integration

1 Introduction

Intelligent tutoring systems (ITS), despite effectiveness as instructional technology, have historically suffered from monolithic design patterns (Murray, 2003). Roschelle and Kaput (1996) referred to tutoring systems as "application islands" for their lack of interoperability. A recent systematic literature review by the author of this paper found little evidence of newer tutoring systems sharing components or working toward a common base of components (Nye, 2013). This lack of modular ITS services reduces the availability of ITS software by preventing sharing of ITS components between systems. This problem increases the cost of ITS development and imposes a high barrier to entry for new systems.

An improvement over this design would be a component-based and service-oriented architecture, allowing composability of ITS components. Composability would greatly benefit ITS research, due to the high interdisciplinary skill-set needed to build a full tutoring system. Service oriented design would allow specialists to focus on individual components, while sharing common components. It would also

greatly reduce the waste of reimplementing components that could be shared by ITS. However, this concept is not new. Roschelle and Kaput (1996) suggested component-based design over a decade ago, but little meaningful progress has been made toward that end. Part of the problem was the relative novelty of tutoring systems: fewer established examples existed and there was less consensus about the definition and functionality of an ITS.

More recently, central researchers have noted that different ITS tools share many of the same high-level behaviors (VanLehn, 2006; Woolf, 2009). This consensus implies a common ontology for describing the high level functions of ITS components and the meaning of information passed between them. While literature consensus does not constitute a formal ontology, it indicates the possibility of a grammar for talking about the types of information communicated between different parts of an ITS. An argument against the feasibility of this approach might be the disconnected nature of many subfields of ITS research, which come from different theoretical backgrounds that are not easily integrated (Pavlik and Toth, 2010). With that said, regardless of the underlying theory, the external behaviors (e.g., giving a hint) and core assessments (e.g., learning gains) are quite similar. The need to maintain theoretical coherence does not mean that a common ontology is infeasible, but simply indicates that there are limits to its useful granularity. For example, does a user-interface care how a hint is generated? If not, the user interface should be able to display hints from any system capable of generating hints. By taking advantage of the distinct roles and functions within a tutoring system, breaking down an individual tutoring system into distinct, sharable components is possible. Moreover, a significant number of components of the tutoring system are secondary to the tutor's theoretical concerns but pivotal to their operation. Machine learning algorithms, data storage interfaces, facial recognition software, speech synthesis, linguistic analysis, graphical interfaces, and tutoring API hooks for 3D worlds are enabling technologies for tutoring systems (Pavlik et al., 2012; Nye et al., 2013).

AutoTutor and the Generalized Intelligent Framework for Tutoring (GIFT) are two separate tutoring frameworks that have independently recognized the importance of modularity and interoperability in tutoring design. AutoTutor is a highly-effective natural language tutoring system where learners talk through domain concepts with an animated agent (Graesser et al., 2004a). Learning gains for AutoTutor average 0.8σ over reading static text materials on the same topic (Graesser et al., 2012). GIFT is a service-oriented framework for integrating tutoring capabilities into static material, such as a PowerPoint, and interactive environments, such as a simulation or a serious game (Sottilare et al., 2012). This paper describes the process of moving AutoTutor toward a service-oriented paradigm and the progress toward integrating AutoTutor with GIFT.

2 Prior AutoTutor Design Patterns

The original AutoTutor design was implemented as a standalone desktop application to teach computer literacy, which also relied on platform-dependent elements such as the Microsoft Agent (Peter Wiemer-Hastings et al., 1998). Since an installed applica-

tion made AutoTutor harder to deliver, a subsequent version reimplemented the tutoring system as a web-based application (Graesser et al., 2004a). Since that time, various tutoring systems that followed in AutoTutor's footsteps have used a mixture of desktop and web-based designs. While many of these systems share conceptual principles and some share authoring tools, reuse of components and services between these different tutoring projects has been limited. So then, while Roschelle and Kaput (1996) spoke of "application islands," AutoTutor and related systems have evolved as a sort of "application archipelago" of related but independent tutoring systems. While the principles of AutoTutor have been influential, code reuse has been limited, even in projects that explicitly extend AutoTutor, such as AutoTutor Lite (Hu et al., 2009).

AutoTutor's package that handles linguistic analysis is a counter-example to this pattern. Coh-Metrix provides a suite of linguistic analysis tools, such as latent semantic analysis, regular expression matching, and domain corpora (Graesser et al., 2004b). While this tool started development nearly a decade ago, it remains under active development and is used regularly by AutoTutor and other projects. This longevity may be attributed to its focused scope and purpose as a toolbox for linguistic analysis. Additionally, Coh-Metrix has the advantage that it is primarily algorithmic and algorithms do not tend to change much.

By comparison, the landscape of educational computing has changed greatly over that period: web-based applications replaced many desktop applications, then full-featured Java web applications were replaced by lighter JavaScript and Flash clients with server-side code written in languages such as Python and C#. AutoTutor designs have mirrored these trends fairly closely, with the original AutoTutor written as a desktop application (Peter Wiemer-Hastings et al., 1998), the next iteration being a Java-based web application (Graesser et al., 2004a), and systems such as AutoTutor Lite relying on Flash, JavaScript, and Python (Hu et al., 2009). In the process of changing platforms and programming languages, a great deal of development work has been lost to a cycle of re-implementation to match the needs of a changing technology landscape.

Based on this history, how could design patterns be improved to encourage reuse and interoperability? The first principle, demonstrated by Coh-Metrix, is embodied by the Unix philosophy: "Do one thing and do it well" (Raymond, 2003). This is fundamental to service-oriented design, where boundaries between components are strict. The second principle is that delivery platforms may evolve rapidly. Just as AutoTutor has adapted to web delivery for desktops, mobile applications are becoming an important platform. Tutoring systems need to minimize platform-dependence. Finally, the best programming languages for different platforms vary. Moreover, existing tutoring systems have large investments in their code base. Components need to communicate using language-agnostic standards for different tutoring systems to interoperate. Service-oriented designs, while not yet common in tutoring systems, offer significant advantages for the next generation of ITS.

3 Sharable Knowledge Objects

AutoTutor is moving in this direction with Sharable Knowledge Objects (SKO), which allow creating tutoring modules by composing a mixture of components: local

static media, remote static media, local components, and web services. These components are categorized in terms of two questions: 1. Is the component local? and 2. Is the component static or interactive? While the current focus of this work is on service-oriented web delivery, the design is also intended to support communication between components in the same process. By using a uniform messaging pattern, components can be developed without consideration of whether they will be used on a local device or accessed as a remote service.

In design pattern terms, SKO's are being developed to follow the service composition principle. In service composition, a composition of multiple services can be considered a single service when creating a new composition of services. Service-oriented design is largely the same concept as component-based design, except with the added complexity that the components may be distributed across time and space as part of a distributed network. So then, what is a SKO? A SKO declares a composition of services intended to deliver knowledge to a user, with the expected use case being tutoring in natural language. In this context, the SKO framework is not a re-implementation of AutoTutor but a framework for breaking AutoTutor down into minimal components that can be composed to create tutoring modules that may or may not rely on the traditional AutoTutor modules. These minimal components are intended to be used as part of a service-oriented design.

Figure 1 shows an overview of the new SKO framework. The core of the new SKO framework relies on a publish-and-subscribe architecture based entirely on passing messages that convey semantically-tagged information. These patterns significantly improve the flexibility of service composition for tutoring. Publish and subscribe frees individual components from explicit knowledge of any other services. The component knows only its own state, the messages that it has received, and the messages that it has transmitted. SKO is viewed as a way to split AutoTutor into separate, easily-reusable components. Secondly, SKO is intended to unify components from different systems that have evolved from AutoTutor along divergent paths by adding their unique functionality as services.

Exploring the details of each of these services is outside the scope of this paper. Instead, this section will focus on how different users would interact with and benefit from a SKO. While certain features of SKO are still under development, these examples describe how different users will interact with the completed SKO framework. To the learner, a SKO acts as a single module of instructional content focusing on a single lesson (e.g., learning how to complete a given math problem). For AutoTutor Lite, a web page loads a talking head and a user-input box, often with a button to begin a tutoring session. The SKO module does not specify any rules or functions. Instead, it relies on components to send messages. So then, user input triggers on the tutoring button generates a message from the user interface component. The tutoring engine reads that message and selects tutoring dialog, which is sent off as a new message. The animated agent and text-to-speech services read this message and cause the talking head to speak the message to the learner. By sharing a student model in a learning management system, multiple SKO can be combined into larger lesson units.

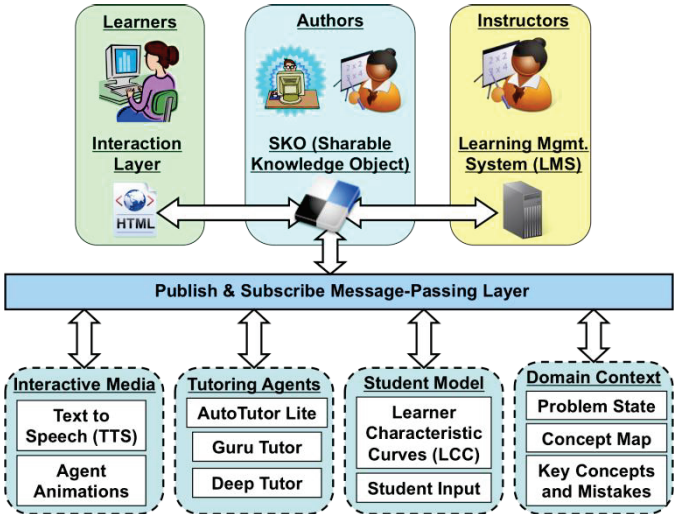

Fig. 6. Sharable Knowledge Object Framework for AutoTutor

To an advanced developer, a SKO is a collection of services. Advanced developers design new services and create SKO templates that can be filled in by instructors. These designers can create a SKO template using an advanced interface, where they would define the set of services within a SKO template and how these tie into the user interface. However, the advanced developer is not expected to add any domain content. Instead, they merely specify placeholders for content that is required or allowed. Based on these placeholders, a form-based authoring wizard would be created to allow instructors and domain experts to create specific SKO based on the template.

To an instructor, a SKO is a series of forms where they enter their expert data and produce working tutoring modules that they can test immediately. For example, an advanced developer could make a SKO template for guiding a student through solving an Algebra problem. From this, a form would be generated to allow an instructor to specify solution steps and tutoring dialogs associated with each step. An instructor could complete this form multiple times to enter content for different problems. This development is intended to be collaborative. By storing SKO in cloud hosts, different authors can edit or test each module. This also greatly facilitates SKO delivery, as a web-based SKO can be directly tested after creation.

4 Integration with GIFT

As part of the project to integrate AutoTutor with GIFT, AutoTutor Lite is being broken down into distinct services to fit into the SKO framework. Rather than focus on the low-level details of how AutoTutor and GIFT are integrating, the high-level process will be outlined. There is no canonical set of services that a given tutoring system should be broken down into so that it can be integrated into GIFT. However, the general integration process followed by AutoTutor might serve as a model for other sys-

tems considering GIFT integration. This integration has five phases: 1. identifying complementary functionality, 2. determining distinct "parts" of the AutoTutor Lite system, 3. specifying the functionality, inputs, and outputs of each part, 4. building web services, and 5. working with GIFT developers to add these to the GIFT distribution.

In the first phase, to identify complementary functionality between GIFT and AutoTutor, a large table of various key features for each system was created. This table helped identify the tools that GIFT had already implemented and those that AutoTutor Lite could contribute. This process identified that AutoTutor's main contributions were conversational pedagogical agents, interactive tutoring, improved student modeling, and semantic analysis tools to compare sentence similarity. In the second phase, the full AutoTutor Lite system was examined to find distinct parts: sets of functionality that could be meaningfully split into distinct components. GIFT is meant to be a generalized system, so re-usable components offer more value to the system. To find these divisions, we looked for parts that only needed and returned small, well-formed information from other parts (e.g., the semantic service can compare any two sets of words and return a similarity value). In the next phase, the functions, inputs, and outputs of each part were determined. After that, we started building web services for each part. Web services were used because they follow communication standards that mean that AutoTutor code does not need to be in the same language as the GIFT code, nor does it need to run on the same computer. Finally, as versions of these web services have been completed, they have been provided to GIFT for integration into the system. This is an important part of the process, as testing with GIFT has helped uncover hurdles about the scalability and limitations of these new services. As these services are completed, they are being integrated into releases of GIFT.

Overall, integration with GIFT dovetails with a larger movement of AutoTutor toward a service-oriented architecture. This redesign will not only help integration with GIFT, but also with other systems in the future. Figure 2 shows how AutoTutor services are expected to integrate into the GIFT framework. AutoTutor services are shown on the right side of the diagram and include the semantic analysis service (for analyzing user input), learner's characteristic curve (LCC) service (a simple type of student model), tutoring service for AutoTutor Lite, a service for text-to-speech, and an animated agent service. Some of these components are already available as web services. Once these services are available, GIFT will be able to incorporate basic AutoTutor Lite tutoring as part of its framework. The message-passing SKO framework will then standardize how AutoTutor communicates with GIFT. Additional services not displayed are also anticipated, such as a persistent student model, authentication service, and services for wrapping assessments such as multiple choice tests.

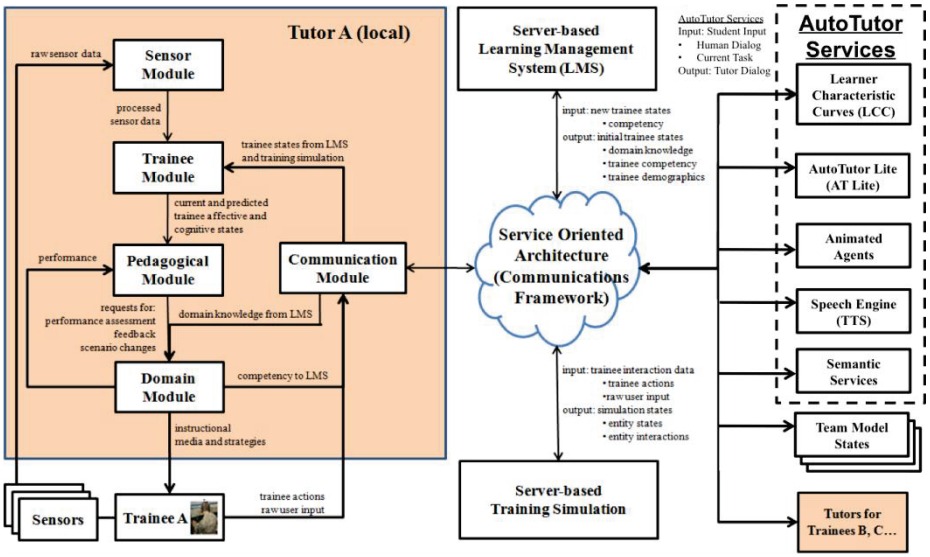

Fig. 7. Integration of AutoTutor and GIFT

5 Limitations and Future Directions

The SKO framework is intended to separate components based on the knowledge transferred between them, represented as semantic messages. This process will greatly improve modularity, enable AutoTutor to be implemented using a service-oriented design, and support interoperability with GIFT. However, modularity is limited by the information each component must share. Certain functions of the tutoring system are more easily separated into distinct components than others. For interoperating with additional tutoring systems, agreeing on a common set of messages may also be a challenge.

Currently, the publish-and-subscribe version Sharable Knowledge Object framework is under active development. In parallel with this work, AutoTutor Lite is being broken down into services and consumed by GIFT using traditional API's. Work in this area is focused on converting the semantic analysis services and AutoTutor Lite tutoring interpreter into services. Message-passing interfaces will then be incorporated into each service and they will be composed using the publish-and-subscribe SKO framework.

Acknowledgements The core SKO architecture is sponsored by the Office of Naval Research STEM Grand Challenge. Integration of AutoTutor Lite services with GIFT is sponsored by the Army Research Lab.

6 References

1. Graesser, A.C., Conley, M.W., Olney, A.: Intelligent tutoring systems. In: Harris, K.R., Graham, S., Urdan, T., Bus, A.G., Major, S., Swanson, H.L. (eds.) APA Educational psychology handbook, Vol 3: Application to learning and teaching, pp. 451–473. APA, Washington, DC (2012)
2. Graesser, A.C., Lu, S., Jackson, G.T., Mitchell, H.H., Ventura, M., Olney, A., Louwerse, M.M.: AutoTutor: A tutor with dialogue in natural language. Behavior Research Methods, Instruments, and Computers 36(2), 180–192 (2004a)
3. Graesser, A.C., McNamara, D.S., Louwerse, M.M., Cai, Z.: Coh-Metrix: Analysis of text on cohesion and language. Behavior Research Methods, Instruments, and Computers 36(2), 193–202 (May 2004b)
4. Hu, X., Cai, Z., Han, L., Craig, S.D., Wang, T., Graesser, A.C.: AutoTutor Lite. In: AIED 2009. IOS Press, Amsterdam, The Netherlands (2009)
5. Murray, T.: An overview of intelligent tutoring system authoring tools. In: Authoring Tools for Advanced Technology Learning Environments, pp. 493– 546 (2003)
6. Nye, B.D.: ITS and the digital divide: Trends, challenges, and opportunities. In: AIED 2013 (2013)
7. Nye, B.D., Graesser, A.C., Hu, X.: Multimedia learning in intelligent tutoring systems. In: Mayer, R.E. (ed.) Multimedia Learning (3rd Ed.). Cambridge University Press (2013)
8. Pavlik, P.I., Maass, J., Rus, V., Olney, A.M.: Facilitating co-adaptation of technology and education through the creation of an open-source repository of interoperable code. In: ITS 2012. pp. 677–678. Springer, Berlin (2012)
9. Pavlik, P.I., Toth, J.: How to build bridges between intelligent tutoring system subfields of research. In: Aleven, V and Kay, J and Mostow, J. (ed.) ITS 2010. LNCS, vol. 6095, pp. 103–112 (2010)
10. Peter Wiemer-Hastings, Arthur C. Graesser, Derek Harter: The foundations and architecture of AutoTutor. In: Goettl, B.P., Halff, H.M., Redfield, C.L., Shute, V.J. (eds.) ITS 1998. LNCS, vol. 1452, pp. 334–343. Springer, Berlin (Sep 1998)
11. Raymond, E.S.: The Art of UNIX Programming. Addison-Wesley (2003)
12. Roschelle, J., Kaput, J.: Educational software architecture and systemic impact: The promise of component software. Journal of Educational Computing Research 14(3), 217–228 (1996)
13. Sottilare, R.A., Goldberg, B.S., Brawner, K.W., Holden, H.K.: A modular framework to support the authoring and assessment of adaptive computer-based tutoring systems (CBTS). In: I/ITSEC (2012)
14. VanLehn, K.: The behavior of tutoring systems. International Journal of Artificial Intelligence in Education 16(3), 227–265 (2006)
15. Woolf, B.: Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning (2009)

Authors:

Benjamin D. Nye is a post-doctoral fellow at the University of Memphis, working on tutoring systems architectures as part of the ONR STEM Grand Challenge. Ben received his Ph.D. from the University of Pennsylvania and is interested in ITS architectures, educational technology for development, and cognitive agents.