Toward a Generalized Framework for Intelligent Teaching and Learning Systems: The Argument for a Lightweight Multiagent Architecture

Benjamin D. Nye and Donald M. Morrison

Institute for Intelligent Systems, The University of Memphis, Memphis, Tennessee
{bdnye and dmmrrson}@memphis.edu

Abstract The U.S. Army's Generalized Intelligent Framework for Tutoring (GIFT) is an important step on the path toward a loosely coupled, service-oriented system that would promote shareable modules and could underpin multiagent architectures. However, the current version of the system may be "heavier" than it needs to be and not ideal for new or veteran ITS developers. We begin our critique with a discussion of general principles of multiagent architecture and provide a simple example. We then look at the needs of ITS developers and consider features of a general-purpose framework which would encourage message-driven, multiagent designs, sharing of services, and porting of modules across systems. Next, we discuss features of the GIFT framework that we believe might encourage or discourage adoption by the growing ITS community. We end by offering three recommendations for improvement.

1 Introduction

As the term is used in a seminal paper on the subject, "Is it an agent, or just a program?" (Franklin & Graesser, 1997), an autonomous agent is

> *..a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to affect what it senses in the future. (p. 25)*

Because a human is also an agent according to this definition, in a sense any intelligent tutoring system may be considered a multiagent system (MAS), designed to support interactions between two agents—the user and the intelligent tutor. However, recent years have seen an increasing interest in the development of systems with multiagent architectures in the more interesting sense that functionality is decentralized across different software agents. In this paradigm, each agent has its own knowledge base (set of beliefs), and carries out different tasks, either autonomously or at the request of other agents. Agent-oriented services build on component-based approaches by giving each component distinct goals that it works to fulfill. As a result, the intelligent behavior of the system as a whole emerges from the collective behavior of the individual agents—including, of course, the human user—allowing for what

has been called "autonomous cooperation" (Hülsmann, Scholz-Reiter, Freitag, Wucisk, & De Beer, 2006; Windt, Böse, & Philipp, 2005). For recent examples of ITSs that employ multiagent architectures, see Bittencourt et al., 2007; Chen & Mizoguchi, 2004; El Mokhtar En-Naimi, Amami, Boukachour, Person, & Bertelle, 2012; Lavendelis & Grundspenkis, 2009; and Zouhair et al., 2012). Although these are for the most part prototypes, they serve as useful demonstrations of the general approach.

Multiagent architectures depend on a shared agent communication language (ACL) such as Knowledge Query and Manipulation Language (Finin, Fritzson, McKay, & McEntire, 1994), FIPA-ACL (O'Brien & Nicol, 1998), or JADE (Bellifemine, Caire, Poggi, & Rimassa, 2008), all of which are based on speech act theory (Austin, 1965; Searle, 1969). The ACL, combined with a shared ontology (semantic concepts, relationships and constraints), allows the agents to exchange information, to request the performance of a task, and, in certain cases—such as when one agent requests access to restricted data—to deny such requests (Chaib-draa & Dignum, 2002; Kone, Shimazu, & Nakajima, 2000). A multiagent architecture therefore consists of a distributed "society" of agents (Bittencourt et al., 2007), each with its own agenda, semantically-organized knowledge base, and ability to send and receive messages. The messages take the form of speech acts, including *requests*, *directives*, *assertions*, and so forth. Here is an example:

```
request
:receiver pedagogical agent
:sender NLP agent
:ontology electronics
:content (define, capacitor)
```

where the message is clearly identified as a *request*, the receiver is a pedagogical agent, and the sender is a natural language processing (NLP) agent that translates utterances from human language into messages the pedagogical agent can understand. In this case the pedagogical agent can fulfill the request because it has access to an ontology in the domain of electronics, and "knows" how to extract a definition from it, by following an algorithm or production rule. Here's another example:

```
tell
:receiver pedagogical agent
:sender emotion sensor
:ontology learner affect
:content (learner, confused)
```

where the receiver is again a pedagogical agent, but in this case the sender is an emotion sensing agent reporting its belief that the learner is currently confused. Again, the pedagogical agent can process the contents of the message because it has access to a "learner affect" ontology. As a final example, consider the following:

```
tell
:receiver LMS agent
:sender pedagogical agent
```

```
:ontology learningExperiences
:content (learner, "passed", "helicopter simulation training")
```

where in this case the pedagogical agent is the sender, and the receiver is an LMS agent, which is being told that a certain learner has passed a training course.

These simple examples illustrate several important principles regarding the nature and behavior of multiagent systems. First, note that all three of the software agents are capable of autonomous action, in accordance with their own agendas, and without the need for supervision. The pedagogical agent need not ask the emotion sensor to report its estimate of the learner's affective state. Rather, the emotion sensor reports its beliefs automatically and autonomously, as it does for any agent that has subscribed to its services. Similarly, when it has judged that a learner has passed a course, the pedagogical agent informs the LMS agent, again without having to be asked, simply because the LMS agent has subscribed to its services.

These agents are "lightweight" in the sense that their power lies in their ability to exchange messages with other agents, and to process the contents of these messages based on ontologies that are shared with the agents they exchange messages with, but not necessarily by all of the agents in the system. For example, the NLP agent and pedagogical agent must both have access to the *electronics* ontology, and the LMS agent and pedagogical agent must both share the ontology of *learner experiences*, but neither the emotion sensing agent nor the LMS agent need to know anything about electronics.

Note also that, assuming that the agents' messages are sent over the Internet, all four agents (including the learner) can be at different, arbitrary locations, whether on servers or local devices. Also, any agent can be replaced by any other agent that performs the same function and uses a compatible ACL and associated ontology. If an emotion-sensing agent comes along that does a better job than the original, then, so long as it reads and writes the same kinds of messages and has a compatible ontology (e.g., terms can be translated meaningfully from one ontology to the other), the other agents don't need to be reconfigured in any way. Most importantly, the functionality and value of membership in the society for all participants can increase incrementally, perhaps even dramatically, by registering new agents with new capabilities, or by upgrading the capabilities of the existing members.

Transforming a monolithic ITS legacy system into one with a distributed, multiagent architecture requires two steps: breaking apart existing components into agents and developing ACLs with ITS-tailored ontologies. By encouraging ITS developers to reorganize their systems as services, the Generalized Framework for Intelligent Tutoring (GIFT) provides strong support for this process (Sottilare, Goldberg, Brawner, & Holden, 2012).

2 Criteria for a MAS ITS Framework

Before discussing GIFT specifically, general criteria required for an effective multiagent ITS framework will be discussed. To understand the criteria for a development framework, one must understand something about the stakeholders involved. In this case, as we are focusing on the software development practices of an ITS, these stakeholders are the research groups that develop these systems. So then, what do

such groups look like? A recently completed systematic literature review of papers including the terms “intelligent tutoring system” or “intelligent tutoring systems” found that the majority of ITS research was split between two types: major ITS families (those with 10 or more papers in a 4-year period) and single-publication projects (Nye, 2013). Together, these account for over 70% of ITS research with each accounting for a fairly equal share. This means two things. First, any generalized framework should be able to accommodate major ITS projects that have a large prior investment in tools. Second, it means that such a framework should also embrace contributions from new developers who are often focused heavily on only a single ITS component (e.g., linguistic analysis, assessment of learning, haptic interfaces). So then, an ideal framework would facilitate breaking down legacy architectures into multiagent systems and would also make it easy for one-off developers to add or replace a single component. The framework should also not be locked-in to a single template for the components included in the system: not all systems can be easily broken down into the same components. However, this walks a fine line: too much structure hinders innovative designs, while too little structure offers little advantage over a generic architecture (e.g., off-the-shelf service-composition frameworks).

Accommodating these different ends of the spectrum requires a lightweight and flexible architecture. However, what do we mean by “lightweight?” There are multiple meanings for a “lightweight framework” and most of them are favorable in this context. The following features can be either lightweight or heavy: (1) hardware requirements, (2) software expertise to design services, (3) software expertise to use existing services, (4) software expertise to stand up the message-passing layer between agents, and (5) minimal working message ontology. The first requirement is that the no special hardware or excessive computational overhead should be required to use the framework. The computational requirements should be light, rather than imposing heavy overhead or unnecessary inter-process or remote service calls. Components requiring significant setup or maintenance (e.g., databases, web-servers) should be optional or, at a minimum, streamlined with default setups that work out of the box.

Assuming self-interest, for both types of developers (veterans and newcomers), the cost of designing or redesigning for the framework would need to be exceeded by the benefits. This means minimizing development overhead to create new services or refit old services for the framework. The generalized framework would need to allow easy wrapping or replacement of existing designs, rather than forcing developers to maintain two parallel versions of their ITS. Researchers and developers are unlikely to develop for a framework that requires extensive additional work to integrate with. This means that new developers should need to know only the minimal amount of information about the framework in order to integrate with it. There should be little to no work to create a simple service that can interoperate with the framework and default wrappers should exist for multiple programming languages to parse raw messages into native objects. Such wrappers or premade interfaces would allow even relatively “heavy” communication between agents, while keeping developers from needing to know these protocols.

The framework must also make it easy to take advantage of services that others have implemented, such as through a repository of publically-available services. At

minimum, it should be significantly easier to use existing services than it is to add a module to the system. This means that the minimal use case (e.g., the "Hello world" case) for the system should be very simple. For example, a single installed package should make it possible to author (or copy) a single text file configuring the system to create a basic ITS test system. Anything required to run a basic example beyond these requirements indicates a "heavier" setup requirement to begin using the framework. If this part of the framework is heavy, first-time ITS authors would be unlikely to use the framework. Moreover, without such ease-of-use, established ITS developers would be unlikely to rework their code to fit such a framework unless they were compensated for these efforts. In the long term, the success and survival of a general framework for tutoring relies on its ability to contribute back to the ITS community. If researchers and developers benefit by reusing services in the system, they will use it. Otherwise, it will fall into obscurity.

Standing up message passing coordination must be lightweight as well. This means that developers should need to expend minimal effort to invoke a layer capable of exchanging messages between services. As such, this layer should have a strong set of defaults to handle common cases and should work out of the box. Additionally, it should be possible to invoke this layer as part of a standalone application (message passing in a single process) or as a remote web service. Consideration must also be given to mobile devices, as mobile applications have specific limitations with respect to their installation, sandboxing (access to other applications), and data transmission.

Finally, agent communication relies on specific messaging languages codified explicitly or implicitly. Three major paradigms are possible to control this communication. The oldest and most traditional paradigm defines API function interfaces for various types of agents or agent functionality, where "messages" are technically function calls on agents. This approach, however, is fragile and better-suited for synchronous local communication than for asynchronous distributed agents. The second paradigm is to define a centrally-defined ontology of messages, which each having an agreed-upon meaning. The main advantage of this system is that it imposes consistency: all agents can communicate using this predefined ontology. However, agreeing upon a specific ontology of messages is an extremely hard task in practice. This approach is "heavy" from the perspective of learning and being constrained by the ontology. The ultimate goal of a shared and stable ontology for ITS is valuable, but offers formidable pragmatic challenges. The third paradigm allows ad-hoc ontologies of messages. At face value, this approach seems flimsy: the ontology of messages is not defined by the agent communication language and services can define their own messages that may not be meaningful to other services as a result. However, this approach is actually fairly popular in research on agent communication languages (Li & Kokar, 2013) and in recent standards bodies, such as the Tin Can API associated with SCORM (Poltrack, Hruska, Johnson, & Haag, 2012). These approaches standardize the format of messages (e.g., how they are structured) but not the content. Instead, certain recommendations for tags and messages are presented but not required. This approach is lightweight: only a small ontology is required and developers are free to extend it.

Lightweight ad-hoc message ontologies show the most promise for an ITS framework using agent message passing. By standardizing the message format, any two services can syntactically understand any message passed to it. However, it al-

lows developers to choose any set of messages for their agent communication language. While in theory this could lead to a Babylon of disjointed ontologies, in practice developers will typically attempt to use established formats for messages first, if they are available. Much like the original design of a computer keyboard or choice of which side of the road to drive on, the starting ontology for a framework can provide a powerful self-reinforcing norm that guides influences work. As such, it is possible to define a core set of suggested messages that are used by the initial set of agents designed for the framework. Additional messages could then be added to the "common core ontology" of messages when they became common practice among new agents added to the service.

3 The GIFT Framework as a MAS ITS

Given these five characteristics, we now look at how well the GIFT architecture matches them in its current form. First, it must be noted that the intentions of the GIFT project are both ambitious and admirable: without the general shift toward service-oriented design for ITSs there would be little value in discussing multiagent ITSs that build upon service-oriented principles. However, this analysis finds that the current implementation of GIFT appears heavier than would be ideal for the needs and practices of ITS developers. This does not mean that GIFT is a bad architecture, simply that it is an architecture that is geared toward the needs of stakeholders other than existing ITS developers (e.g., end-users, sponsors, etc). A great deal of emphasis is placed on reliability and stability, which is more reflective of enterprise use rather than rapid development. The current GIFT implementation implies a "consume and curate" service model rather than a "collaborative repository" service model. With help from GIFT experts, it is certainly possible to integrate tutoring services with GIFT and deliver this tutoring effectively using the architecture. However, the architecture does not seem light enough to allow researchers to build it into their own toolchain. This section first examines the strengths of GIFT as a generalized framework for developing tutoring systems and then considers limitations that might be addressed by future releases.

By far, the primary advantage over existing systems is its dedication to service-oriented principles and modular design. GIFT is the first serious attempt to develop a platform intended to inject a common suite of tutoring services into a variety of applications, including web applications and 3D games (Sottilare, Goldberg, Brawner, & Holden, 2012). GIFT also has a strong commitment to standards-based communication protocols, supporting the Java Messaging Service (JMS) for service communication. Finally, GIFT was developed in Java so it can be efficiently interpreted on web servers and has strong cross-platform capabilities. The hardware requirements for the core GIFT system are also light. Modern systems should have no trouble running the GIFT services and communication layer. Overall, GIFT appears to be well-optimized for efficient delivery and hosting of tutoring web services.

However, the current GIFT implementation has significant limitations as a development framework for tutoring systems. First, the current implementation does not offer an easy road for standing up a minimal working example using the GIFT

framework. Installing and setting up the core framework for use is a multi-step process with multiple stages and some third-party dependencies. Running the framework also requires setting up a dedicated database, which could never be considered a light feature. While some GIFT ITS may benefit from such a database (e.g., those hosting surveys), many prototype ITS might make do with simpler triple-stores, serial data (e.g., delimited text files), or even no persistent data storage. Additionally, setting up the GIFT framework does not differentiate between the core architecture meant to handle communication between services versus the services that are bundled with the architecture. A barebones version might remedy this limitation. Services also communicate using a classical API paradigm, which does not offer much flexibility compared to a more explicit message-passing approach. This means that a developer would need to inspect individual service interfaces to figure out the appropriate accessors. Effectively, this locks developers into an ontology of how services should *act* (i.e., remote API requests) rather than what they should *know* (e.g., generic beliefs or knowledge). While this may seem like a subtle difference, a service that only needs to broadcast its knowledge can sidestep designing who receives that information and how it should be used. Finally, GIFT lacks service stubs or wrappers in common languages (e.g., Java, Python, C#) that would make it easy to develop a service that conforms to the framework.

Overall, deploying the GIFT architecture and attempting to develop a new service for the system are both heavy tasks rather than lightweight ones. Without support from the GIFT project, this would make developing for the framework quite costly. The software expertise to design services is heavy, since there are few tools to make this process easier. Despite using a service-oriented paradigm, the system does not offer a suite of example services or stub service in common programming languages. Unless developers have expertise in Java and can carefully inspect the available API, they would not be able to integrate a new service into GIFT. The software expertise to use existing services is also heavy. The minimal use-case example currently installs all GIFT services and requires a database. Services are not handled using a repository or package manager approach, but are simply installed with no streamlined method to manage them. Since there is no way to install the service communication layer as a standalone system, the software expertise to stand up any message-passing layer between agents is also heavy. Finally, no message ontology is available because the system messages are invoked to carry API calls between services. While ontologies for GIFT have been discussed, these ontologies are focused on the types of services in the system rather than the types of messages employed (Sottilare, 2012). This forces communication between services to revolve around the API of services rather than on the information they are passing.

In its current form, the GIFT framework would not be well-suited for a multiagent system ITS. It also does not support many of the aspects of such a framework that would aid either of the major classes of ITS developers to base their projects on GIFT. A one-off innovation, such as a PhD candidate's thesis project, would likely be significantly burdened by the effort to stand up the system without help and would need to learn the API for existing services before they could be useful. A large group focusing on an established ITS architecture would be limited by these factors and also by the lack of interfaces and supporting tools for the programming languages used by their legacy projects. Most importantly, since services do not communicate

using a more general agent communication language, significant effort will likely be required to tailor communication to the specific API function interfaces. Without the ability to specify a common message ontology for the agent communication language, it would be impractical to develop a multiagent tutoring system using GIFT. Traditional API's based on interfaces are not well-suited to this task, as they conflate process names with the meanings of the data they produce. Traditional API functions are also poorly suited for dynamic function binding and other advanced patterns that could be used by message-passing agents.

4 Discussion and Recommendations

This analysis has explored the potential benefits and requirements related to building an intelligent tutoring system based on multiagent architecture principles and an agent communication language. These requirements were then compared with the GIFT framework's current capabilities. Our finding is that the current implementation of GIFT is not currently well-suited to these advanced design patterns. While hardware requirements are low, software expertise to design new GIFT services and to use the existing GIFT services is fairly high. Additionally, message system of GIFT currently reflects an API pattern with heavy reliance on knowing the other services in the framework. This is unfortunate, as lighter publish-and-subscribe patterns have become increasingly popular in the industry due to their adaptability (Jokela, Zahemszky, Rothenberg, Arianfar, & Nikander, 2009). This said, GIFT represents a project that is far closer to these patterns than any prior ITS project. GIFT has also spurred discussion on patterns for service-oriented tutoring that were not previously at the forefront of ITS design.

Based on this analysis of GIFT, some design recommendations are indicated for future iterations. From the perspective of developing tutoring agents, the first major recommendation is to center communication of services around explicit message passing where agents publish their knowledge using speech acts. To support this goal, feedback should be gathered from major ITS research groups to propose messages for an initial ontology of recommended messages that determine the information passed between components of the system. To add services to the GIFT framework, developers should only need to know this ontology of messages so they can use it or extend it accordingly. Services should not need to know who their messages are received by, only what messages they receive, what messages they produce, and when they wish to produce a message.

The second major recommendation is the need to separate the GIFT services from the GIFT communication layer. If GIFT is truly a general framework, it must ultimately provide a specialized communication layer as its core. Other services should be treated as plug-ins that can be installed or removed using a package-management approach. This includes the core GIFT services that are bundled with the system. Separating the services from the core architecture would greatly simply the ability to provide a minimal working example and would make the system more flexible overall. As the system itself appears to be designed with such boundaries in mind, this should primarily be a matter of how setup packages are structured and installed.

Related to this issue, a very basic installation that works "out of the box" must be available for developers to start working with GIFT.

The third major recommendation is that GIFT should provide small suites of utilities, wrappers, and stubs to help develop services using a variety of common languages. A generalized system must not assume that developers will convert their code to Java or build their own communication wrappers for their native language. While the use of remote procedure API calls has sidestepped this issue slightly, it has not completely removed it. Additionally, a more flexible message-passing paradigm would require such supporting tools to an even greater extent.

Finally, in the present analysis we have focused only on issues of system architecture, as is proper given that GIFT intends to serve as a general purpose framework, not a stand-alone ITS. However, in so doing we have arguably paid insufficient attention to other important issues that GIFT approaches, such as the need for shareable domain models, learner models, and instructional modules. As the developers of GIFT have pointed out, legacy ITSs tend to be built as "unique, one-of-a-kind, largely domain-dependent solutions focused on a single pedagogical strategy" (Sottilare, Brawner, Goldberg & Holden, 2012:1). After some four decades of independent effort, a case can be made that the time has come for a much greater degree of collaboration and sharing among members of the ITS community, including both veterans and newcomers. This means not just the sharing of ideas, but of working software objects and structures. The development of a lightweight, multiagent architecture that supports "autonomous cooperation" among communities of distributed software agents united by an emergent common language offers a first step in the process, but it is by no means the last.

5 References

1. Austin, J. L.: How to do things with words. Oxford University Press, New York (1965)
2. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE: A software framework for developing multi-agent applications. Lessons learned, Information and Software Technology, 50(1), 10–21 (2008)
3. Bittencourt, I. I., de Barros Costa, E., Almeida, H. D., Fonseca, B., Maia, G., Calado, I., Silva, A. D.: Towards an ontology-based framework for building multiagent intelligent tutoring systems. In: Simpósio Brasileiro de Engenharia De Software. Workshop on Software Engineering for Agent-oriented Systems, III, João Pessoa, 2007. Proceedings of the Porto Alegre, SBC, pp. 53–64 (2007)
4. Chaib-draa, B., Dignum, F.: Trends in agent communication language. Computational Intelligence, 18(2), 89–101 (2002)
5. Chen, W., Mizoguchi, R.: Learner model ontology and learner model agent. Cognitive Support for Learning-Imagining the Unknown, 189–200 (2004)
6. El Mokhtar En-Naimi, A. Z., Amami, B., Boukachour, H., Person, P., Bertelle, C.: Intelligent Tutoring Systems Based on the Multi-Agent Systems (ITS-MAS): The Dynamic and Incremental Case-Based Reasoning (DICBR) Paradigm. IJCSI International Journal of Computer Science Issues 9(6), 112–121 (2012)

7. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the third international conference on Information and knowledge management, pp. 456–463. ACM (1994, November)
8. Franklin, S., Graesser, A.: Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In: Intelligent agents III agent theories, architectures, and languages, pp. 21–35. Springer Berlin Heidelberg (1997)
9. Hülsmann, M., Scholz-Reiter, B., Freitag, M., Wucisk, C., De Beer, C.: Autonomous cooperation as a method to cope with complexity and dynamics?–A simulation based analyses and measurement concept approach. In: Y. Bar-Yam (ed.), Proceedings of the International Conference on Complex Systems (ICCS 2006), vol. 2006. Boston, MA, USA (2006)
10. Jokela, P., Zahemszky, A., Rothenberg, C., Arianfar, S., Nikander, P.: LIPSIN: Line speed publish/subscribe inter-networking. In: ACM SIGCOMM Computer Communication Review, 39(4), pp. 195–206. ACM Press (2009, August)
11. Kone, M. T., Shimazu, A., Nakajima, T.: The state of the art in agent communication languages. Knowledge and Information Systems, 2(3), 259–284 (2000)
12. Lavendelis, E., Grundspenkis, J.: Design of multi-agent based intelligent tutoring systems. Scientific Journal of Riga Technical University. Computer Sciences, 38(38), 48–59 (2009)
13. Li, S., Kokar, M. M.: Agent Communication Language. In: Flexible Adaptation in Cognitive Radios, pp. 37–44. Springer New York (2013)
14. Nye, B. D.: ITS and the Digital Divide: Trends, Challenges, and Opportunities. In: Artificial Intelligence in Education (In Press).
15. O'Brien, P. D., Nicol, R. C.: FIPA–towards a standard for software agents. BT Technology Journal, 16(3), 51–59 (1998)
16. Poltrack, J., Hruska, N., Johnson, A., Haag, J.: The Next Generation of SCORM: Innovation for the Global Force. In: The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC), vol. 2012, no. 1. National Training Systems Association (2012, January)
17. Searle, J. R.: Speech acts: An essay in the philosophy of language. Cambridge University Press (1969)
18. Sottilare, R. A.: Making a case for machine perception of trainee affect to aid learning and performance in embedded virtual simulations. In: Proceedings of the NATO HFM-169 Research Workshop on the Human Dimensions of Embedded Virtual Simulation. Orlando, Florida (2009, October)
19. Sottilare, R. A.: Considerations in the development of an ontology for a generalized intelligent framework for tutoring. In: Proceedings of the International Defense and Homeland Security Simulation Workshop (2012)
20. 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: The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC), vol. 2012, no. 1. National Training Systems Association (2012, January)
21. Windt, K., Böse, F., Philipp, T.: Criteria and application of autonomous cooperating logistic processes. In: Gao, J.X., Baxter, D.I., Sackett, P.J. (eds.) Proceedings of the 3rd International Conference on Manufacturing Research. Advances in Manufacturing Technology and Management (2005)
22. Zouhair, A., En-Naimi, E. M., Amami, B., Boukachour, H., Person, P., Bertelle, C.: Intelligent tutoring systems founded on the multi-agent incremental dynamic case based reasoning. In: Information Science and Technology (CIST), 2012 Colloquium, pp. 74–79. IEEE (2012, October)

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.

Dr. Chip Morrison is a Faculty Affiliate at IIS. A graduate of Dartmouth, Dr. Morrison holds an M.A. from the University of Hong Kong and an Ed.D. from Harvard. His current research interests include models of human cognition and learning, and the application of these models to conversation-based intelligent learning systems.