University of South Carolina
Scholar Commons

Faculty Publications

Computer Science and Engineering, Department of

2005

Service-Oriented Computing: Key Concepts and Principles

Michael N. Huhns
University of South Carolina - Columbia, huhns@sc.edu

Munindar P. Singh

Follow this and additional works at: https://scholarcommons.sc.edu/csce_facpub

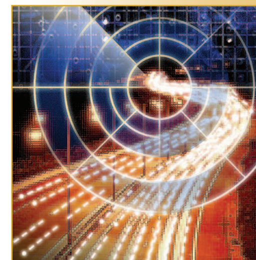
Part of the Computer Engineering Commons

Publication Info

Published in *IEEE Internet Computing*, ed. Michael N. Huhns and Munindar P. Singh, Volume 9, Issue 1, 2005, pages 75-81.
http://ieeexplore.ieee.org/servlet/opac?punumber=4236
© 2005 by the Institute of Electrical and Electronics Engineers (IEEE)

This Article is brought to you by the Computer Science and Engineering, Department of at Scholar Commons. It has been accepted for inclusion in Faculty Publications by an authorized administrator of Scholar Commons. For more information, please contact digres@mailbox.sc.edu.

Editors: Michael Huhns • *huhns@sc.edu*
Munindar P. Singh • *singh@ncsu.edu*



Service-Oriented Computing: Key Concepts and Principles

Traditional approaches to software development — the ones embodied in CASE tools and modeling frameworks — are appropriate for building individual software components, but they are not designed to face the challenges of open environments. Service-oriented computing provides a way to create a new architecture that reflects components' trends toward autonomy and heterogeneity.

Michael N. Huhns
University of South Carolina

Munindar P. Singh
North Carolina State University

Although the Web was initially intended for human use, most experts agree that it will have to evolve — probably through the design and deployment of modular services — to better support automated use. Services provide higher-level abstractions for organizing applications for large-scale, open environments. Thus, they help us implement and configure software applications in a manner that improves productivity and application quality.

Because services are simply a means for building distributed applications, we cannot talk about them without talking about service-based applications — specifically, how these applications are built and how services should function together within them. The applications will use services by *composing* or putting them together. An architecture for service-based applications has three main parts: a provider, a consumer, and a registry. Providers publish or announce their services on registries, where consumers find and then invoke them.

Current Web service standards and techniques support these parts and enable many important use cases, but the particular set of basic standards they employ are incidental to the key concepts underlying service-oriented computing (SOC).[1] Indeed, although Web services provide a ready source of practical examples, they are unnecessarily limited. The architecture for Web services provides a framework that can be fleshed out with more powerful representations and techniques taken from established computer science approaches. Serious practitioners already use such representations, although they are omitted from most expositions of Web services. The articles in this track will thus emphasize SOC concepts instead of how to deploy Web services in accord with current standards. To begin the series, we describe the key concepts and abstractions of SOC and the elements of a corresponding engineering methodology.

 1089-7801/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society

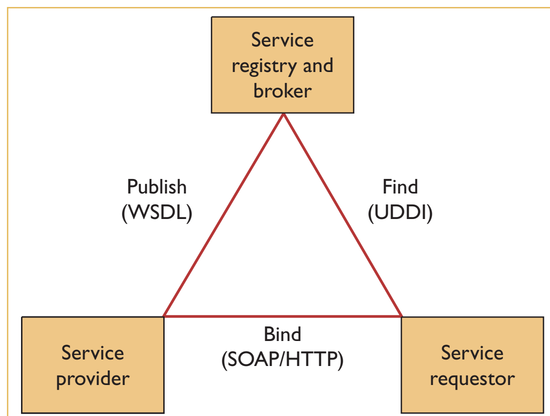


Figure 1. Web services architectural model. As a basis for a service-oriented architecture, Web services models incorporate how Web services are advertised, discovered, selected, and used.

Abstractions for SOC

We can view SOC in terms of several different cross-cutting levels of abstractions, ranging from those that concern services within an application to those that concern service applications interacting across enterprises. Let's consider a typical surgery division in a large hospital. One challenge in such a setting would be to make the payroll, scheduling, and billing systems interoperate. Each system is quite complex, with its own user interfaces and databases, perhaps running on different operating systems.

For obvious reasons, these systems must work together. Scheduling employees and operating rooms for surgery is complicated, for example, because schedules require frequent updating. A scheduling system must balance staff and equipment availability with unpredictable levels of surgical urgency and advance notice.

The mechanisms for payroll are similarly complex – the payroll system must consider various kinds of overtime rules for different categories of labor, such as nurses, residents, consulting physicians, senior surgeons, radiologists, and so on – and rely to some extent on data from the scheduling system.

Likewise, the billing system must also incorporate scheduling information. It is used not only to bill customers, but also to deal with medical insurance companies and government agencies (such as those for children, the elderly, retired government employees, and veterans). Agencies typically impose complex rules for valid billing and penalties for violations of these rules. Across the US, for example, different rules apply regarding how hospitals can bill for anesthesiologists' efforts during surgeries. In some states, a senior anesthesiologist can supervise up to four junior (resident) anesthesiologists. This senior person thus time-shares his or her effort among four concurrent surgeries, whereas each junior person is dedicated to one. If an emergency arises, the law allows a fifth surgical procedure under the same senior anesthesiologist's supervision, but the billing rate is severely reduced.

Hospital systems provide services within an enterprise, exemplifying the abstraction level of intraenterprise interoperation. Similar examples can exist among enterprises, among software components, and between the application level of a system and its infrastructure level, as the following sections describe.

Intraenterprise Abstraction Level

If we look at just the interoperation aspects in our hospital example, we quickly see several hurdles to overcome. The first is connectivity among the applications, which protocols such as HTTP can readily ensure. The second is the ability of the various components to understand each other. XML, particularly XML Schema, can handle communication formatting, but it cannot decipher the meaning behind a given message. Meaning usually is not encoded explicitly; rather, it depends on how various systems process information, which means system integrators and developers must uncover and reconcile the interacting components' intent. This reconciliation presupposes that accurate declarative information models exist. In practice, however, such models are often poorly maintained or simply do not exist. Developers must thus construct (or reconstruct) them at integration time. This process is further complicated by the fact that information models for different systems might describe different abstraction levels. Services can encapsulate component behavior at many levels, but still describe it in the same way, thus easing composition of the components.

Because our hospital setting deals with legacy and other systems and applications that operate within the enterprise, various enterprise policies must readily authenticate and authorize the parties involved in different interactions. A service-oriented architecture (SOA), by requiring that policies be made explicit, can organizationally enforce compliance with these policies, thus simplifying the system's management.

A new problem arises when we want to introduce new applications and configure them to interoperate. Imagine that the hospital purchases an anesthesia information management system (AIMS) to complement its existing systems. An AIMS helps anesthesiologists manage procedures in surgery as well as monitor, record, and report activity such as various gas and drip levels. This information establishes compliance with government regulations, ensures that certain clinical guidelines are met, and supports studies of patient outcomes. Although it is easy to buy an AIMS, installing and using it is much more challenging. To introduce one in the operating room, the installer must ensure that the right interface is exposed between the new and existing systems — systems likely developed on different platforms and perhaps running on different operating systems. Clearly, adherence to interconnection standards is crucial.

Even with low-level connectivity, problems with messaging (components connecting operationally) and semantics (components understanding each other) remain. We also have to configure and customize the new application's behavior: an AIMS must be populated with a hospital-specific data model, so that it displays the correct user interface screens to the staff and logs the right observations.

SOC addresses these problems by providing the abstractions and tools to model the information and relate the models, construct processes over the systems, assert and guarantee transactional properties, add flexible decision-support, and relate the functioning of the component software systems to the organizations that they represent.

Interenterprise Abstraction Level

Traditionally, enterprises have interoperated in an ad hoc manner that required substantial human intervention. Alternatively, they use rigid standards such as Electronic Data Interchange (EDI), which leads to difficult-to-maintain systems. Recently, we have seen a growing interest in supply-chain management and flexible, on-demand manufacturing, which has led, in turn, to more cross-enterprise processes in general. The idea is that businesses that have to work together anyway can improve their responses to information, reduce overhead, satisfy individual customer preferences, and exploit emerging opportunities by streamlining their interactions through technology.

Returning to our hospital scenario — specifically, the billing portion — illustrates this process. The traditional approach would be to send hardcopy bills to agencies and insurance companies, which the receiving party would then retype into its information system. Naturally, such approaches are disappearing in favor of online billing.

Yet, online systems must capture data formats in a reliable manner so that insurance companies can understand hospital-formatted information, and vice versa. If a standard approach existed, robust commercial tools could process the format rather than the custom-software-based systems that are still widespread. In recent years, industry has converged on XML as the data format of choice. Although it is clearly a success, it does not resolve how communicated data is to be understood and processed.[2]

Next, imagine that the hospital wants to buy catheters. To efficiently purchase this supply, the hospital must interoperate with a catheter vendor, but let's further suppose the hospital wants to use whichever vendor offers the best terms. This type of *dynamic selection* is increasingly common as people recognize the benefits of its flexibility. If an entity can pick its business partners flexibly, it can select them to optimize any kind of quality-of-service criteria, including performance, availability, reliability, and trustworthiness.

Suppose a hospital were performing a business transaction with a supplier and encountered an error, such as the supplier being temporarily out of stock. It would be great if the hospital could rewire the interaction to an alternative supplier dynamically and transparently to the overall process. To do this, the hospital's purchasing system would need some means of recovery to restore a consistent state and restart the computation with new suppliers.

SOC provides the ability for interacting parties to choreograph their behaviors, so that each can apply its local policies autonomously, yet achieve effective and coherent cross-enterprise processes. In addition, it provides support for dynamic selection of partners as well as abstractions through which the state of a business transaction can be captured and flexibly manipulated; in this way, dynamic selection is exploited to yield *application-level fault tolerance*.

Infrastructure Abstraction Level

Building complex applications over distributed platforms, such as grid architectures, is difficult and has led to an interest in modular interfaces based on services. Accordingly, the grid research community views grid services as analogous to Web services.[3]

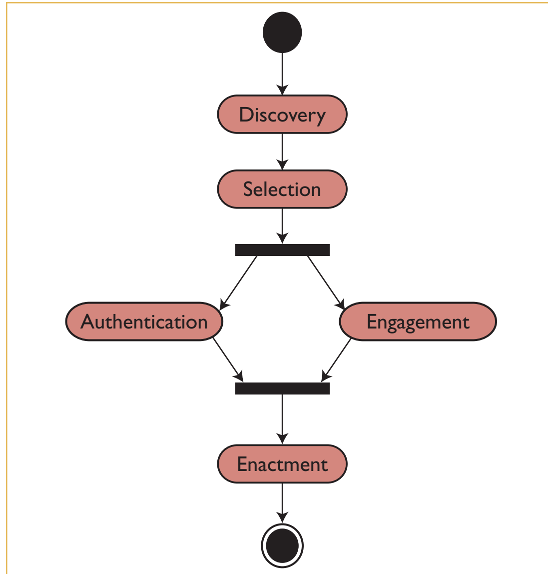


Figure 2. Normative workflow. For the development and execution of a Semantic Web service in a service-oriented architecture, an appropriate methodology must support the development of each state shown.

A Grid provides computing resources as a utility analogous to electric power or telecommunications, so that enterprises can concentrate on their core business and outsource their computing infrastructure to specialist companies. IBM and Hewlett-Packard, for example, currently offer utility computing services.[4]

Utility computing presupposes that diverse computational resources can be brought together on demand and that computations can be realized depending on demand and service load. In other words, utility computing assumes service instances would be created on the fly and automatically bound as applications are configured dynamically. A service viewpoint abstracts the infrastructure level of an application. It enables the efficient usage of grid resources and facilitates utility computing, especially when redundant services can be used to achieve fault tolerance.

Software Component Abstraction Level

Programming abstractions that consider software components to be potentially autonomous help improve software development. Services offer programming abstractions in which software developers can create different software modules through interfaces with clearer semantics. When a programmer uses the full complement of semantic representations for services, the resulting modules are more easily customizable. The result is that SOC provides a semantically rich and flexible computational model that can simplify software development.

Service-Oriented Architectures

The hospital use case provides a challenging set of requirements for any approach to computing, but an SOA can satisfy them all. Figure 1 provides a general architectural model for Web services, which form the basis for an SOA.

SOC's emphasis falls on the architecture, because many of the key techniques for its components — databases, transactions, software design — are already well understood in isolation. Practical success depends on how well we can place these techniques into a cohesive framework, so that we can apply them in production software development. Recent progress on standards and tools is extremely encouraging in this regard.[5,6]

Several SOAs can coexist, provided they satisfy some key elements for SOC:

- *Loose coupling.* Tight transactional properties — maintaining and guaranteeing data and state consistency — generally do not apply among components because conventional software architectures do not typically include transaction managers. Although it would not be appropriate to specify data consistency across the various components' information resources because they are autonomous, we must consider the high-level contractual relationships that specify component interactions to achieve system-level consistency.
- *Implementation neutrality.* The interface for each component matters most, because we cannot depend on the interacting components' implementation details, which can be unique. In particular, a service-based approach cannot be specific to a set of programming languages, which cuts into the freedom of different implementers and rules out the inclusion of most legacy applications.
- *Flexible configurability.* An SOC system is configured late and flexibly, which means different components are bound to each other late in the process. Thus, the configuration can change dynamically as needed and without loss of correctness.
- *Persistence.* Services do not necessarily require a long lifetime, but because we are dealing with computations among autonomous heterogeneous parties in dynamic environments, we

must always be able to handle exceptions. The services must exist long enough to detect any relevant exceptions, take corrective action, and then respond to the corrective actions taken by others. They should also exist long enough to engender trust in their behavior, because they are engaged dynamically and reputation might be the only means available to gauge their reliability. Ephemeral services would not be around long enough to develop a reputation.

- *Granularity.* An SOA's participants should be modeled and understood at a coarse granularity. Instead of modeling interactions at a detailed level, we should capture the high-level qualities that are (or should be) visible for business contracts among the participants. Coarse granularity reduces dependencies among participants and reduces communications to fewer messages of greater significance.
- *Teams.* Rather than framing computations centrally, we should think in terms of how autonomous parties, working on a team as business partners, realize those computations. Researchers in multiagent systems confronted the challenges of open systems early on, when they attempted to develop autonomous agents that could solve problems cooperatively or compete intelligently.[7]

Although SOAs might not be new, they address the fundamental challenges of open systems, which are to operate efficiently and achieve coherence in the face of component autonomy and heterogeneity. SOC adds the ability to build on conventional information technology in a standardized way, so that tools can facilitate the practical development of large-scale systems.

Engineering an SOA

Because an SOA differs from a conventional architecture, it requires a different development methodology than CASE tools typically provide. A popular and typical methodology for creating conventional software — the unified modeling methodology (UMM) — involves first describing business collaborations as use-case diagrams and then describing business transactions as activity diagrams. CASE tools then generate the code to implement the system.

As shown in Figure 2, a service-oriented methodology replaces code generation with a combination of service discovery, selection, and engagement. Some or all of these steps might occur at runtime.

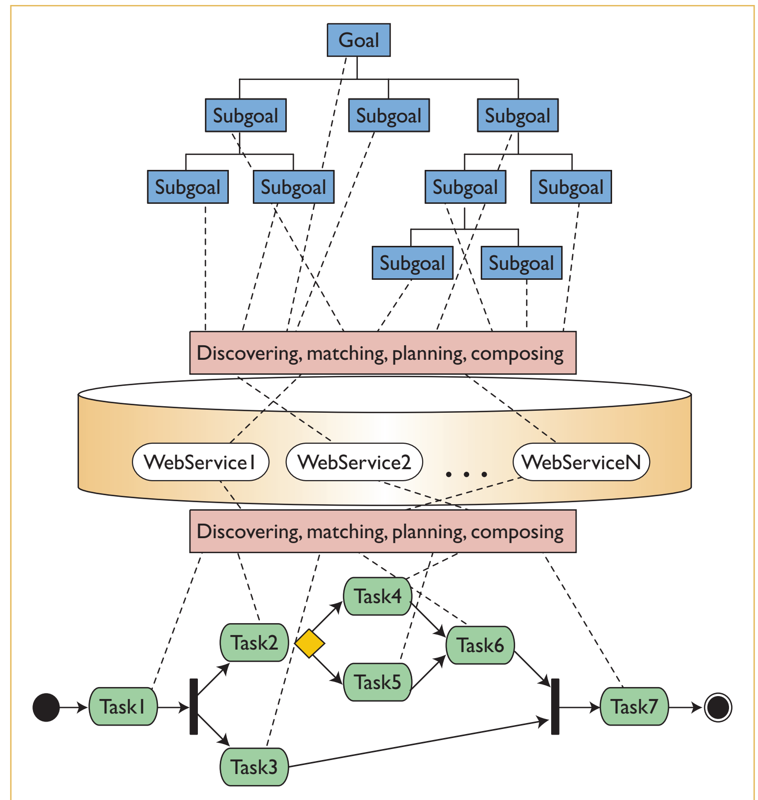


Figure 3. Service-oriented architecture. Engineering a service-oriented computing system is a process of discovering and composing the proper services to satisfy a specification, whether it is expressed in terms of a goal graph (top), a workflow (bottom), or some other model.

Figure 3 presents a different view of what is involved in engineering an SOA. In this case, service composition plays a key role.

Service composition is crucial because it lets us create new value from existing parts. Reuse is a well-regarded concept in traditional software development, but it is merely a convenience, whereas reuse is essential in the case of services, because services cut across organizational boundaries. In traditional software, for example, you can implement your own data structure or graphics package, but you cannot implement your own insurance provider or airline. You have no choice but to deal with other people's services, so you must be able to put them together — or compose them — appropriately.

Composite services apply in many practical settings. Portals, for example, aggregate information from multiple sources: the challenge is to personalize the information for each user. Electronic commerce can involve aggregating product bundles to meet specific user needs. Virtual enterprises and

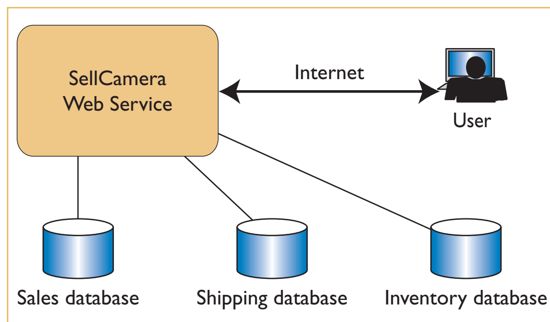


Figure 4. Business-to-consumer transaction environment. This camera-selling Web site must at times try to infer a user's state and intentions, and in doing so, it adopts agent-like behaviors.

supply-chain management reflect generalizations of consumer-oriented e-commerce scenarios, because they include subtle constraints among a larger number of participants.

Challenges for Composition

Consider a simple business-to-consumer (B2C) situation, as depicted in Figure 4: a company sells digital cameras over the Web, combining an online catalog with up-to-date models and prices, a valid credit-card transaction process, and a guaranteed delivery. The B2C transaction software would

- record a sale in a sales database;
- debit the credit card;
- send an order to the shipping department;
- receive an approval from the shipping department for next-day delivery; and
- update an inventory database.

However, problems can arise at several points in the process. What if the order ships, but the debit fails? What if the debit succeeds, but the order is never entered or shipped? In a closed environment, transaction-processing (TP) monitors can ensure that all or none of the steps are completed and that the systems eventually reaches a consistent state, but such TP monitors do not apply over the Web. Suppose the user's modem were disconnected right after he or she clicked on OK. Did the order succeed? Suppose the line went dead before the acknowledgment arrived. Would the user order again? The basic problem is that a TP monitor cannot get the user — part of the software system's open environment — into a consistent state. The TP monitor cannot control any part of the environment outside its scope.

There are several possible solutions for handling issues such as these in open environments.

- The server application could send email about credit problems or detect duplicate transactions.
- A downloaded Java applet could synchronize with the server after the broken connection was reestablished and recover the transaction; the applet could communicate using HTTP or directly with server objects via Corba's Internet Inter-ORB Protocol or remote method invocation (RMI).
- To make the processing robust against demand fluctuations, orders could be put in a message queue, which could be managed by *message-oriented middleware*. Because MOM guarantees message delivery or failure notification, customers would be notified by email when the transaction was complete.

People regularly use email to communicate with each other, so in using email, the server behaves like an intelligent agent, further exemplifying the emerging agent-like aspects of the Web and its services.

Although our camera store example considers a user dealing with a particular enterprise, the problem is more acute in business-to-business settings. If our camera store were considered merely a component in a large supply network, it would have no hope of forcing the other parties to conduct their local transactions in any particular manner or to reliably converge to a consistent state across the system. Deeper models of transactions and business processes could ensure that the correct behavior was realized in such cases.

Current specifications for Web services do not address transactions or specify a transaction model. Emerging standards include the Business Transaction Protocol[8] as well as IBM, Microsoft, and BEA's WS-BusinessActivity and allied standards (http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-BusinessActivity.pdf). However, most implementers believe that SOAP will manage transactions — somehow. Without guidance from a standard or an agreed-upon methodology by the major vendors, transactions will be implemented in an ad hoc fashion, thus unnecessarily complicating interoperability and extensibility.

Security will also be more difficult because more participants will be involved, and service designers might not anticipate the nature of all the participants' interactions and needs. We will probably also see incompatibilities in vocabularies, semantics, and pragmatics among service

providers, brokers, and requesters. Performance and QoS issues might provide additional challenges.

Spirit of the Approach

Although simplistic, Figure 1 makes apparent many of the problems that must be solved for an SOA to become viable on a large scale. To publish effectively, we must be able to specify services with precision and greater structure. Parties who are not from the same administrative space as the provider will eventually invoke the service, and differences in assumptions about the service's semantics could be devastating. The registry must be able to certify the given providers so that it can endorse them to the registry's users.

Service requestors should then be able to find a registry that they can trust, which means dealing with considerations of trust, reputation, incentives for registries, and, most importantly, for the registry to understand the requestor's needs. Once a service is selected, the requestor and the provider must develop a finer-grained sharing of representations. They must be able to participate in conversations to conduct long-lived flexible transactions, in such a manner that they can establish and monitor a service-level agreement.

The keys to the next-generation Web are cooperative services, systemic trust, and an overall understanding based on semantics. The agent characteristics of proaction and autonomy, and agents' ability to negotiate commitments and deal with exceptions, are needed for the anticipated applications of SOC. Research is required to enable a transition to the next generation, especially because the Web presents a demanding environment for applications. In particular, its size and dynamism present problems for open applications, but its size fortuitously provides a means for solving them as well. A given topic, for example, might contain an overload of information — much of it redundant and some of it inaccurate — but a system can use a preponderance of evidence or voting among the different information sources to reduce the information to that which is consistent and agreed upon. Competing service providers might not be trustworthy, but a system could use a reputation network to assess credibility. Subsequent articles in this track will address these and other concepts, further exploring and formalizing the principles expressed throughout this article.

Acknowledgments

This article is based on *Service-Oriented Computing: Semantics, Processes, Agents* (John Wiley & Sons, 2005).[1] Parts of that text were reproduced with permission.

References

1. M.P. Singh and M.N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, John Wiley & Sons, 2005.
2. D.L. McGuinness and F. van Harmelen, eds., "OWL Web Ontology Language Overview," World Wide Web Consortium (W3C) recommendation, Feb. 2004; www.w3.org/TR/owl-features.
3. I. Foster et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration," Globus, 17 Feb. 2002; www.globus.org/research/papers/ogsa.pdf.
4. J.W. Ross and G. Westerman, "Preparing for Utility Computing: The Role of IT Architecture and Relationship Management," *IBM Systems J.*, vol. 43, no. 1, 2004, pp. 5–19.
5. F. Curbera et al., "Unraveling the Web Services Web," *IEEE Internet Computing*, vol. 6, no. 2, 2002, pp. 86–93.
6. R. Khalaf et al., "Understanding Web Services," *Practical Handbook of Internet Computing*, M. Singh, ed., Chapman Hall CRC Press, 2005, chapter 27.
7. F. Bergenti, M.-P. Gleizes, and F. Zambonelli, eds., *Methodologies and Software Engineering for Agent Systems: The Handbook of Agent-Oriented Software Engineering*, Kluwer Academic, 2004.
8. S. Dalal et al., "Coordinating Business Transactions on the Web," *IEEE Internet Computing*, vol. 7, no. 1, 2003, pp. 30–39.

Michael N. Huhns is a professor of computer science and engineering at the University of South Carolina, where he also directs the Center for Information Technology. His research interests are in the areas of SOC, multiagent systems, enterprise integration, and ontologies. Huhns received a BS in electrical engineering from the University of Michigan, Ann Arbor, and an MS and a PhD in electrical engineering from the University of Southern California. He is a member of the IEEE, the ACM, the AAAI, Tau Beta Pi, Eta Kappa Nu, and Sigma Xi. Contact him at huhns@sc.edu.

Munindar P. Singh is a professor of computer science at North Carolina State University, where he dabbles in agents and services, and especially business protocols, trust, and social networks. Singh has a BTech in computer science and engineering from the Indian Institute of Technology and a PhD in computer sciences from the University of Texas. He's also authored close to 200 articles and columns, and he served as editor-in-chief of *IEEE Internet Computing* from 1998 to 2002. Contact him at singh@ncsu.edu.