From the Editor-in-Chief . . .

Engineering Interoperation

Munindar P. Singh • singh@ncsu.edu

The Web connects services (information resources, vendors, brokers, and such) to users and to each other. Users want related services to work together—that is, to interoperate. Interoperation differs from integration, which requires multiple services to be combined into a single logical service. While a unified conceptual view is desirable, integration is essentially an old-world concept. On the Web, where the services are heterogeneous, dynamically changing, and autonomous, integration can be messy, fragile, or impossible—depending on the extent of our ambitions.

By contrast, interoperation requires no more than arms-length relationships among services, so it is the more viable alternative. Unfortunately, while programmers often successfully achieve interoperation, their task is rarely pleasant, and their solutions are seldom elegant. This is because current programming models are ill-suited to building systems of components that implement heterogeneous, dynamically changing, autonomous services. To engineer interoperation, we need a programming model to specify it and a behavior model to make it operational.

On the Web, integration can be messy, fragile, or impossible—depending on the extent of our ambitions.

Component Interactions

Software components interact in various ways, and their interactions can be desirable or undesirable. For example, they may deadlock, feed others their results, or wait for others to release resources. Sharing results and resources is usually desirable, but deadlock isn't. More importantly, however, interactions can be involuntary for at least one of the participants, or they can be voluntary (autonomy-preserving) for all. Communications are the latter kind, because the participants choose whether and when to speak and whether to pay heed when spoken to.

Preserving autonomy is where an agent-based approach can help. A major distinction between agents and objects is that objects must do as they are told, while agents needn't comply. When you invoke a method on an object, it had better compute. But you don't invoke a method on an agent, you communicate with it: you request it to do something for you, and it may accept, deny, modify, or simply ignore your request.

Requirements for Interoperation

Given that communication is the right way to achieve interoperation among autonomous components, what requirements should the communications meet so that interoperation can be *engineered*—that is, specified and robustly achieved with reduced development effort?

For autonomous components to interact and function in an unpredictable world, they should be prepared to handle exceptions, and exception handling is where current approaches break down. Handling exceptions presupposes the ability to try a task more than once and in different ways. This requirement forces components to be persistent and possibly nonterminating. When components are long-lived, other components can't wait for them to terminate. This means that components must release some results early. And if they do, they must also be able to revise those results in response to changes in their environment, exceptions they encounter, and even revisions made by other components. Thus, communications must be structured so that components can naturally interact as described above.

Programming Model

Meeting these interoperation requirements calls for a programming model that includes a small interface consisting of patterns of interaction among interoperating components. The following are the more fundamental patterns:

- entertain (specified kinds of) requests from other components,
- notify others of specified information when it becomes available, and
- renotify others when a specified condition is achieved (or, equivalently, violated).

These patterns embody enough flexibility to accommodate many practical situations. However, some additional power would be required for cases of greater complexity. To this end, a component may participate in additional interactions; for example, it may attempt to satisfy application-level constraints and jointly (with other components) attempt to satisfy some mutual constraints.

A component may choose which patterns it will support and for which other components. By voluntarily committing to specific instantiated patterns, the components enter into customized contracts that make for interoperation in settings with autonomous services.

Operational Characterization

Underlying services, especially if preexisting, would generally not include the capability to support the above interface. Consequently, the component or agent responsible for a service would have to implement the necessary interface. Because the patterns mentioned above are process oriented, accommodating them presupposes a sufficiently expressive behavior model for the components. Interestingly, the required behavior model can be captured as a statechart (a well-known software engineering formalism). Statecharts provide a natural operational characterization for the patterns, which makes it easier to put them into practice.

The proposed programming model distills a lot of prior work on multiagent interaction and workflow management.[1] What I like about it is that it is conceptually clean, combines well with traditional software engineering, and offers a natural implementation of high-level abstractions.[2] ■

REFERENCES

1. M.N. Huhns, “Agent Teams Building and Implementing Software,” *IEEE Internet Computing*, Vol. 4, No. 1, Jan. 2000, pp. 93-95.
2. F. Wan et al., “Handling Semantic Exceptions in the Large,” in *Agent-Oriented Information Systems*, G. Wagner and E. Yu, eds., MIT Press, Cambridge, Mass., to appear.

IEEE INTERNET COMPUTING

IEEE Computer Society Publications Office
10662 Los Vaqueros Circle, PO Box 3014
Los Alamitos, CA 90720-1314