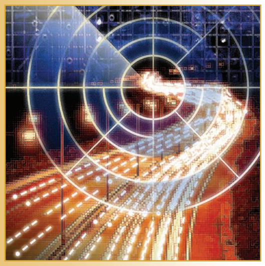

Service Agents and Virtual Enterprises: A Survey

Implementing specific principles of academic software agents could help us build on Web service standards and finally realize the promise of a virtual enterprise.

Charles Petrie
Stanford University

Christoph Bussler
Digital Enterprise Research Institute

In a virtual enterprise (VE), a company assembles a temporary consortium of partners and services for a certain purpose. This purpose could be a temporary special request, an ongoing goal to fulfill orders, or an attempt to take advantage of a new resource or market niche. The general rationale for forming the VE is to reduce costs and time to market while increasing flexibility and access to new markets and resources. As much as possible, individual companies focus on core competencies and mission-critical operations, outsourcing everything else.

One of the ideas driving VE creation is that of processes dynamically constructed out of available Internet-based services as needed at runtime. In the late 1980s, Marty Tenenbaum talked about a "sea of services" on the Internet that would facilitate VE formation. Now that we have Web services, this idea of finding services at runtime has great potential.

Although the VE seems increasingly closer to realization as we move through sequences of Internet technologies and formats, it still remains just out of range. In this article, we examine some of the technical reasons for this fact and suggest the work that remains to be done. We advocate the introduction of academic software agent principles, but we propose abandoning specific implementations in favor of building on emerging Web service standards.

Supply Chains and the Virtual Enterprise

A key step toward achieving a VE is to create a set of standards and conventions that lets software automatically find partners, markets, and services as needed and then integrate them without prior agreement. Such autonomic software is essential for scaling; companies could use it to leverage a global infrastructure that could respond quickly to changing conditions or to form multiple-company special projects that last over widely varying time periods.

Useful Information for Virtual Enterprises

This article describes many concepts and ideas toward realizing a VE. The following URLs describe products and technologies that are leading the way:

- Business Process Execution Language for Web Services, www-106.ibm.com/developerworks/library/ws-bpel
- Business Process Modeling Language, www.bpmi.org/bpml.esp
- Business Transaction Protocol 1.0, www.oasis-open.org/committees/business-transactions
- Collaxa, www.collaxa.com
- Content-based Router, www-ksl.stanford.edu/knowledge-sharing/agents.html
- DAML-S, www.daml.org/services
- EbXML, www.ebxml.org
- EDI — UN/EDIFACT, www.unece.org/trade/untdid/welcome.htm
- FX-Agents, http://fxagents.stanford.edu
- IATA 1998, http://snrc.stanford.edu/~petrie/agents/agent-ec
- Multi-Agent Negotiation Testbed, www.cs.umn.edu/magnet
- Process XML, http://snrc.stanford.edu/~petrie/fx-agents/xserv/pxml.html
- RosettaNet, www.rosettanet.org
- Service-Oriented Negotiation, www.ecs.soton.ac.uk/~nrj/so-neg.html
- Semantic Matchmaker, www-2.cs.cmu.edu/~softagents/daml_Mmaker/daml-s_matchmaker.htm
- Semantic Web-Enabled Web Services, http://swws.semanticweb.org
- Dollar Rent-A-Car case, http://groups.haas.berkeley.edu/citm/conferences/020612/presentations/Segev+patankar.pdf; www.microsoft.com/resources/casestudies/CaseStudy.asp?CaseStudyID=11626
- TAP, http://tap.stanford.edu
- TIBCO, www.tibco.com/solutions/products/
- Universal Business Language (UBL), www.oasis-open.org/committees/ubl/
- UDDI, www.uddi.org/specification.html
- Web Services Conversation Language 1.0, www.w3.org/TR/wscl10
- Web Service Choreography Interface, wwws.sun.com/software/xml/developers/wsci
- Web Services Description Language 1.1, www.w3.org/TR/wsdl/
- Web Services Flow Language 1.0, www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
- Web Services Inspection Language 1.0, www-106.ibm.com/developerworks/webservices/library/ws-wsilspec.html
- XLANG, www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
- XML/EDI, www.geocities.com/WallStreet/Floor/5815
- Yodlee, www.yodlee.com

Consider the abstract scenario of supply chains. A supplier can be a consumer at any point in the process. In response to a request for a quote from supplier 1, for example, supplier 2 could in turn request quotes from supplier 3 and supplier 4. Anticipating these quotes, supplier 2 could give supplier 1 a preliminary quote, but it might need to reneg on the quote later based on supplier 3's and supplier 4's quotes. We call this a recursive supply network (RSN).

RSNs in particular, and VEs in general, must be able to handle contingencies and new opportunities. An RSN requires status monitoring and control; its distributed process plan should be continually monitored and changed as necessary. If we could support such a dynamic real-time VE, business supply chains would be faster and more efficient, and we could complete projects faster and cheaper. Moreover, we could extend this vision to personal travel, dinner, and sport plans, ordering (and changing or canceling) various business services in the process.

Instead of moving toward this vision, however, companies typically have used remote procedure calls (RPC) and private networks based on prearranged contracts. Suppose a company's composite service includes shipping, and it discovers a shipper with superior service for the immediate application. Shouldn't the supply process immediately take advantage of this efficiency? Such a concept can help us imagine completely new ways of doing business — for instance, with third-party brokers like today's auction Web sites, which don't require end-point contracts.

Since Tenenbaum's proposal 20 years ago, protocols that allow screen scraping have commercialized the Internet. Companies specializing in screen scraping in certain domains could sell their metaservices to that domain service's consumers. Web services, especially those built on WSDL and SOAP,[1] could eliminate such intermediaries and move us further toward realizing the VE.

Web Services

Our use of the term *Web services* represents a way of publishing an explicit, machine-readable, common standard description of how to use a service and access it via another program using a standard message transport. Industry has strongly embraced the use of SOAP and WSDL, which have become major standards for Web services. WSDL, in particular, has been accepted quickly — in part because of its simplicity in comparison with more advanced systemic standards such as ebXML.

The importance of SOAP and WSDL is that they offer the possibility of a simple industrial standard for reading what input/output messages a service accepts and sends, and for sending those messages over a standard transport. This loose coupling means that the kind of client or server software at either end is irrelevant. SOAP and WSDL provide an API-like abstraction from software in a lightweight format. Additionally, they are simple open standards with plenty of available tools, most of which are useful for other purposes and are becoming de facto standards in themselves. In contrast, EDI and similar systems, which are perhaps better-designed business-interchange standards, require expertise and special tools as well as much longer construction times.

However, there is a strong risk of disenchantment with WSDL. The object-oriented community has already pointed out some of its problems.[2] Moreover, there is an expectation that we soon will be able to discover and assemble services into new composites dynamically and on the fly, as we need them.

Dynamic Service Discovery with WSDL

How exactly do we discover a Web service automatically and on the fly? Let's examine the discovery of new services, which is usually considered the function of universal description, discovery, and integration (UDDI).

Why Isn't UDDI Enough?

The goal of any service directory, including UDDI, is to enable automatic search for desired services. UDDI also purports to provide sufficient information for using previously unknown services,[3] just as XML was previously touted as enabling the understanding of previously unencountered data and information.[4] However, UDDI does not provide service descriptions, even in theory; it is structured to provide meta metadata about services.

Even in theory, UDDI support for automated search is severely restricted. An official UDDI registry comes with a default set of taxonomies to which other taxonomies can be added (or registered). Suppose our VE is not a travel agency but that it occasionally needs to perform logistics as part of its operations and that we want to book a flight. We would like our software automatically to go to a UDDI — to IBM's UDDI at https://uddi.ibm.com/ubr/registry.html, for example — and ask the computer equivalent of, "are there any Web services that book flights?"

To use the `Find` function, the discovery program would have to query a UDDI node for a set of taxonomies. You can try this yourself by clicking on `Find` and using the *Advanced Search* to *Find a Service*. Taxonomies are not machine-readable because there is no standard syntax (much less semantics) for them. Suppose we had a software agent that we somehow programmed to read new taxonomies, that could select a Web service taxonomy, and that could then use it for selection. This agent could thus send a query to a UDDI node asking for Web services classified via this taxonomy.

But services are not classified by their WSDL operations. We cannot ask for a service that books flights, for example, because service descriptions are stored only in pointers (not in UDDI). Moreover, because UDDI does not provide search functionality over these distributed service descriptions, we can't do any more than discover which services have been registered with a given taxonomy. Although this is possibly a good design decision, it means that UDDI falls short in the semantic description of services necessary for automated search.

In this case, the problem is passed over to WSDL; our software agent can only look for "travel" services. In the same IBM UDDI, look for a `Service Name` starting with "Travel" using `Locator Category` (Taxonomy) "`UNSPC`" and leave the other values blank. The first service name returned will be "Travel Adventures Unlimited." Clicking on this name gets you a page showing the UDDI registry entry. Click on `Details` and *Access Point Address* to get a list of WSDL service operations, which is not informative. (If you haven't followed along, all the operation names are similar to "`P3Typex`.") Clicking on `P3Type3` will get you a Web page interface to the SOAP message showing you that this operation has to do with `airlineID`. A software agent would go directly to the WSDL (click on "Service Description" to see this) to discover that the part name for the *Input Message* for this operation is "`airlineID`," but the expected response is a little mysterious — the output message part name is "body." In fact, none of the operations actually books a flight.

Understanding WSDL

Suppose that UDDI or some other mechanism were to provide distributed search of WSDL descriptions. Could an automated software agent use those descriptions to search for a desired service? The previously described example leads us to suspect not. Let's look at a WSDL example from www.xmethods.com. Suppose we're looking for a service that can find telephone numbers in Sweden. Could we automatically discover the representative service "ISearchSwedishPerson"? Obviously, not with UDDI, but could we do it with an agent that searched this "xmethods" site?

Two problems exist with such a search for a service. One is that this name is not the WSDL service name, which is "IsearchSwedishPersonservice." The more serious problem is that it would take a smart search engine to realize what this service does exactly, especially because the semantics are encapsulated in a C programming style of capitalizing the beginning of otherwise cojoined words.

But let's suppose the program is a good guesser and decides to examine the actual WSDL description at www.marotz.se/scripts/searchperson.exe/wsdl/IsearchSwedishPerson to figure out if this service will serve its intended purpose. This service has five operations:

- `HTMLSearchAddress`
- `HTMLSearchPhone`
- `XMLSearchAddress`
- `XMLSearchPhone`
- `IsAlive`

These names aren't much help because of the naming methodology. Perhaps by examining the I/O messages and drilling down through the XML complex types, software might be able to discover what these operations really mean. For instance, inspecting `operation HTMLSearchAddress`, a program can determine that the Input message name is "`HTMLSearchAddressRequest`." That message, in turn, has part names of

- `fName`
- `lName`
- `Address`
- `ZipCode`
- `City`

If there were reference to a standard taxonomy or ontology (not just those for registering services in UDDI), software would have a pretty good chance of understanding at least the term "City." Type specifications, if they exist, might refer to XML schemas that could give further clues about the messages' semantics, and thus, service operations. That is, something more than just categories of businesses is necessary for service discovery.

Primarily, we'll need a standard taxonomy or ontology of common terms in order to provide semantics for service operations and messages. This could be something as elaborate as DAML-S, TAP, RosettaNet, or even just some informal industry de facto standard terminology. In particular, DAML-S provides a well-designed solution to the problem of providing semantics for distributed search of Web services. While DAML-S is a very important and good solution in many respects, it has two weaknesses. First, it currently only provides information at the service level and not at the WSDL *operation* level. Second, it may be too heavy for industrial requirements. UBL is more likely to be widely adopted as a semantic solution.

Such terminology is more likely to be agreed on at the level of message-part names such as `City`, rather than operation names like `HTMLSearchAddress`. "City" is not defined today in DAML-S, for instance, but it could be. The example shows that discovery programs will have to be smart enough

Primarily, we'll need a standard taxonomy or ontology of common terms in order to provide semantics for service operations and messages.

to parse XML schemas using the taxonomies that provide service operation semantics.

WSDL developers also will have to change their RPC-style thinking. The output message of `HTMLSearchAddress` is `HTMLSearchAddressResponse`, and its only part name is `return`. This unfortunate naming methodology is quite common in WSDL, making it impossible for a program to determine what this Web service is returning; it would have been so much better had the part name been something like `telephone-number`. The part name of another message in another operation is simply `number`. Again, this is too generic to be meaningful to a discovery program unless it is almost as smart as a human. None of this is WSDL's fault. A `return` can easily be defined as a complex XML type that can be parsed automatically by software reading the description. This is a methodological and conceptual problem.

The larger problem is that UDDI passes off to WSDL the issue of having an adequate service description for search; WSDL's designers clearly intended the service directory to solve this problem, so now we're at a stalemate. WSDL descriptions will have to be expanded, and UDDI's functionality will have to include distributed search. WSDL poses a technical problem for any UDDI++

that allows distributed search for WSDL operations. XML conventions allow an operation to be indirectly referenced by a URI, which could be a URL or a global URN. How to extend service descriptions so that UDDI++ could allow a search at the level of operations and messages is an open issue.

Using WSDL to Use a Web Service

If we consider the two WSDL examples described earlier, we see that descriptions sufficient for discovery involve descriptions sufficient for use. How do we know what operations we're looking for if we don't know what the operations actually do? Furthermore, knowing what the operations do presumes knowing how to use them. This is why discovery involves both search and use.

If the search is successful, the service is understood to be applicable for the intended purpose. What might not be understood are the conditions for using the service. Without machine-readable descriptions, a software program cannot really use Web services without a human first reading the Web page descriptions for each service and its operations, and then writing code.

We can illustrate this best by starting from scratch and making an existing Web-based service into a Web service with SOAP messages described by WSDL. In such a case, we do not inherit naming and semantic issues, meaning we can discover higher-level problems with WSDL. In the FX-Agents Project (a collaboration between the Stanford Center for Information Technology, NEC, and Intec Web & Genome to integrate Web technology for financial applications), we chose a restaurant from waiter.com and attempted to reimplement it via SOAP and WSDL. Even in this simple case for a given restaurant, we found that we needed to understand how to execute a sequence of operations in order to use the service: we first had to select a restaurant, get the menu, choose an entree, make selections about the entree, and then execute the regular payment and delivery operations. A UDDI/WSDL++ technology therefore must provide machine-readable instructions for sequencing these service operations.

Over and above this, we found less obvious but important problems with WSDL as a formal representation for dynamically discovered services:

- WSDL handles static sequence specifications, but not *unplanned options*. Pizzas require toppings, for example, but other menu choices do not, meaning such a specification could be handled with a dynamically generated XML schema using import at runtime, although this is not very elegant.
- There is no WSDL commitment to an *authorization* description. In the waiter.com example, there is membership registration with payment. The open question is how to map different authentication mechanisms onto WSDL.
- There is no description of *cancellation* terms. By what hour or day can someone cancel an order, and with what penalties? In the waiter.com example, a three-hour notice is required to avoid a penalty.
- There is no description of service *effects or actions*. How and when will food be shipped? In this example, food ships 90 minutes after order. What messages will be sent? Here, there is a phone call and email later the same day from the restaurant.
- There is no description of service *preconditions*. For instance, with one restaurant, a minimum order is US$80, delivery hours differ from takeout hours, and the restaurant is closed Mondays.
- There is no description of *payment* terms. For some restaurants, there's a delivery charge of $8.95, and a driver-support charge of 15 percent is added to each order.

WSDL defines no message semantics for concepts that might routinely be associated with services, such as reasons for denial of service or even a simple concept like "service-provider." Furthermore, there is no provision for correlating replies — from multiple queries of restaurants, for example, about delivery speed. Important UDDI/WSDL lacunae include the lack of representation trust and level-of-service information.

WSDL technology makes no commitment toward representing these service concepts, so WSDL developers currently have two choices: don't express conditions or express them in an ad hoc manner. Most developers choose the latter option, which works in private practice. If designers choose to express conditions this way, however, they ignore the issues of authorization and payment. Cancellation is an ad hoc operation, but this approach doesn't scale well because every discovery program will have to be somehow programmed for each new payment, authorization, and cancellation operation.

For automatic use, there must also be explicit tags referring to pre- and postconditions for use of the service — for example, Dollar Rent-A-Car requires drivers to be over 18 years of age and

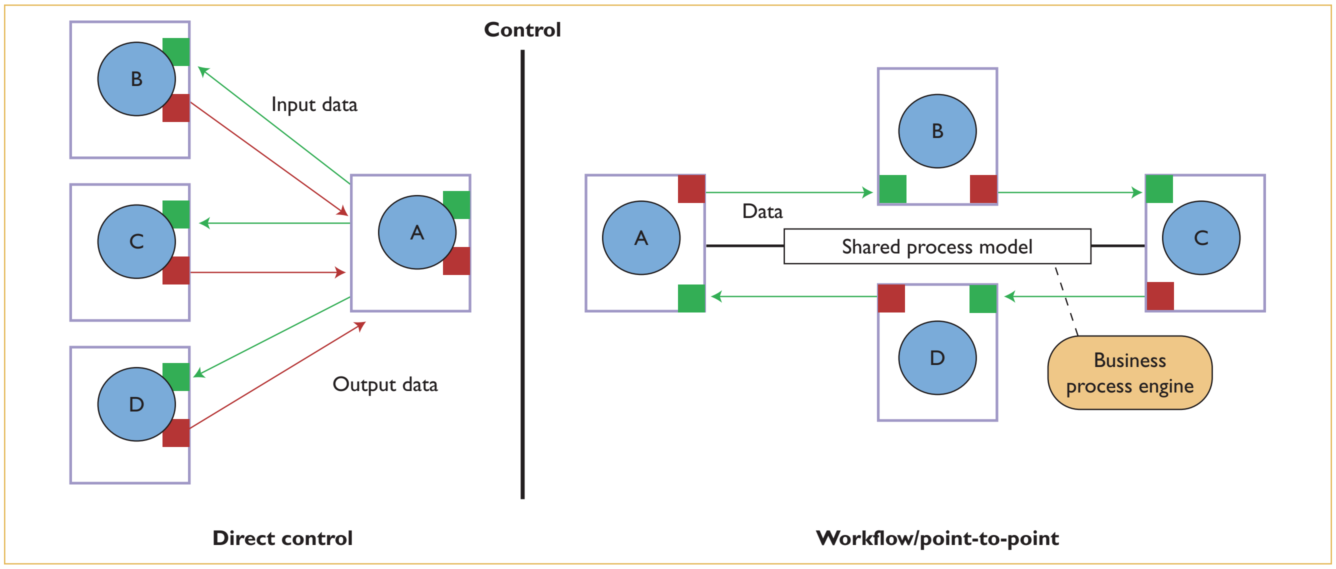

Figure 1. Internal workflow. This figure shows direct control versus control via a shared-process model.

possess a valid driver's license. DAML-S allows service providers to specify such conditions; including this information separately from WSDL could ultimately be the only solution to pre- and postconditions.

WSDL has great promise because it is so simple, but technologies analogous to stylesheets and XML schema could help enrich it. Perhaps a taxonomy will become standardized through the use of WSIL. We suspect the most likely outcome is that particular WSDL operations for special functions such as payment, authentication, and cancellation will become standardized through business implementation and convention, and possibly through the use of UBL.

The lack of representation in WSDL for alternatives, conditions for use of service, and side effects resulting from use of service are extremely serious limitations for engineering VE business processes. These descriptive lacunae are sufficient to prevent the automatic and dynamic use of previously unknown services in a VE and make an RSN practically impossible because new component sources cannot be found on the fly. But the opportunity for interesting research grows by leaps and bounds when we consider Web service integration, which is necessary for VEs.

Dynamic Service Integration

Assuming we have a UDDI/WSDL++ with expressive power sufficient to discover and understand service use dynamically, we would like to integrate some set of services to accomplish a goal, such as making a complex travel plan or RSN. Among our fundamental desiderata for VEs are that any publicly registered Web service can be consumed:

- by anyone, without requiring any change to the service (the *democratic principle*), and
- anytime, without prior arrangement (the *just-in-time principle*).

As Tenenbaum described it, the idea of a "sea of services" is that we can freely consume the service of choice when needed, without requiring the service provider to do anything more than advertise in the public service description used to discover the service.

Service composition is a subissue of integration; essentially, it involves making a set of services into a single visible service. Several systems do an admirable job of providing composition, including WSFL, BPEL4WS, and Self-Serv.[5] Although composition could be an important topic for some VEs, we are most concerned with dynamic service integration, which might not even be compatible with composition.

We can evaluate integration architectures along three dimensions:

- Graph-based versus integration-based control on runtime requirements;
- Point-to-point connections versus messages with receivers and senders decided at runtime; and
- Centralized versus distributed monitoring and control of the process generated by the service integration.

We now examine some alternatives using these principles and dimensions of service integration.

Graph-Based Service Integration

In the simplest case of service integration, a central program using direct control can manage all control, status, and state information. For a given application, such direct control is not a bad solution. It can be accomplished without violating our two fundamental principles.

However, the internal workflow approach is not sufficient if the VE is an RSN. If one of the Web services accomplishes its promised ultimate output by consuming other services, these are completely hidden from the end user and thus can't be monitored, much less controlled. An RSN requires some measure of monitoring and control. One way of achieving this is a workflow-like flow of control, which usually means that a graph of possible transitions among all the services concerned is developed prior to runtime (see Figure 1). This approach violates our fundamental principles to some degree because it doesn't allow the use of services and providers not specified in the graph. However, graph-based architectures vary in the degree to which they allow runtime flexibility. All allow specific service providers to be chosen at runtime from a preexisting set of partners.

Service providers must agree in advance to some shared model — kept and maintained by some entity, and programmed with appropriate endpoint connections.

WxFL models, in particular, assume that the process (composed of service interactions) is just a workflow, that it can be described prior to execution, and that a shared process model can be generated and used by a centralized process engine to control process execution. BPEL4WS and its associated transaction models address some of WSFL's deficiencies but violate both the democratic and just-in-time principles: Web service integration is accomplished via a process graph with end-to-end connections that must be established before execution. More fundamentally, the workflow approach defines specific processes with specific messages among a set of pre-identified partners; it's not based on existing WSDL. Rather, new WSDL code must be written to conform to the process.

This means that service providers must agree in advance to some shared model — kept and maintained by some entity, and programmed with the appropriate endpoint connections. Changing this shared model might mean reprogramming by the service providers, thus making dynamic service discovery, use, and integration for VEs impossible as well as causing maintenance and scalability problems (see the critique of WSxL integration at http://snrc.stanford.edu/~petrie/fx-agents/xserv/icpaper/appendix-d.html).

A point-to-point specification of connections presupposes a graph-based approach. ebXML, BTP, BPML, RosettaNet, and the unimplemented WfMC standard[6] take more of a peer-to-peer transactional approach. However, none of these approaches allow runtime discovery of new actors for roles or new activities, thus violating the just-in-time principle because they assume that all potential partners are identified prior to runtime and that all concrete transitions can be enumerated and described in a process graph prior to runtime. Such standards focus on transactions specified among preidentified partners, rather than generic transactions.

Major vendors have several efforts under way to provide XML-based EDI via SOAP/WSDL, but they could require a central process-execution engine because the EDI standard doesn't constrain the message dialog. Moreover, such systems' general flexibility remains to be seen. The Self-Serv system requires no centralized control engine for a given set of transactions because it generates distributed coordinations at runtime as needed. However, because of its focus on composition, its containers are still graph-based and thus don't allow a completely free choice of services and providers at runtime.

Condition-Based Service Integration

Pre- and postconditions are important to any Web service for operation sequencing, but they're crucial to services that perform actions. Moreover, if we define the preconditions and side effects for each operation, we won't need a graph-like prescription of how to sequence the operations. Software could automatically determine which operations require preconditions or are the effects of other operations. Postconditions partially describe whether the service is accomplished — for example, whether food gets delivered, a flight is booked, or a component is promised for delivery and by when. Preconditions tell us what services might be required to do any of these things; they can include, for example, converting currency, providing components to be shipped for assembly, or reserving a room for the event to be catered.

The idea of connecting services and tasks by pre- and postconditions at runtime is a fundamental function of AI planning[7] as well as simple project management.[8] Condition-based integration can use a workflow graph as a starting point, but it allows for runtime changes. At a specific application level, condition-based integration provides a partial program for service integration, which can be decomposed into goals or constraints that are not satisfied or violated until services are found.[8] Say that our goal is to assemble a PC with certain requirements and that the constraints are to avoid a certain supplier and keep the total cost below a given amount. The software is then free to plan a sequence of services that result in the PC being assembled under those constraints. Many potential plans are possible, so goals and constraints only partially specify the program of service sequences.

Condition-based integration cannot be accomplished by point-to-point connections: it requires peer-to-peer transactions. In addition to the WxFL data messages flowing across the network, monitoring and control messages should coordinate and control the application process instead of the underlying transactions. The WxFL data messages do not implement transactions that provide monitoring and control. Additional messages that provide those functions are needed. In the most flexible case, there would be an Execution Control Language (ECL) similar to the current Agent Communication Languages (ACLs).[9] This approach specifies only generic message types applicable to any process, which potentially allows anyone communicating in the standard set of messages to enter at any time, depending on the authentication enforced. Like an ACL, a peer-to-peer ECL protocol is an abstract partial program for many processes. The ECL protocol is an implicit abstract process model associated with message types; it would determine only whether certain message sequences are legal. (Of course, there also must be further agreement on the messages' data content, based on standard XML schemas.)

In an approach in which we use an ECL for condition-based integration, each business partner is responsible for enforcing protocol, evaluating received messages, and determining whether they are legal. Each is free to send back an ACL `Sorry` message, saying either "I don't understand this message in this context" or "I reject your request."

There is no need for a central process engine with its maintenance and scaling issues. We can still pass data, but it's augmented by a standard for the data's semantics and appropriate actions for that kind of data. For example, data messages might be of types `Purchase-Order`, `Firm-Order`, `Acknowledge-PO`, `Create-Order`, or `Order-Fulfillment`. These message types (along with the shared semantics) would tell the receiver what to do with the data contained in the message, thus adding a basic level of control to the process. Let's call these *action control* message types.

In addition to the control messages already discussed, another flexible set of message types allows dynamic negotiation, which would be useful in an RSN as well as for individual services. Suppose automatic software wanted to negotiate on the user's behalf. How could someone use a discount coupon to get a cheaper rate, or to know to ask what the options are? How would a supplier tentatively agree to terms for delivery, based on other pending agreements? We can more or less directly use other ACL message types for these sorts of situations, such as those of the "contract net" protocol.[10] Extensions could enable a choice of negotiation protocols at runtime. In any case, ECL negotiation primitives for a given protocol would look something like `solicit-bid`, `propose`, `accept/reject`, `commit/reneg`, and so forth. These *negotiation control* message types add another level of control and are already implemented in a commercial solution.[11]

Dynamic Process Management

Although sufficient for dynamic process integration, action and negotiation control messages are not sufficient for the final consumer, or (recursively) for each consumer in the RSN, to predict whether the process goals will be met on time or if action should be taken to correct the process. An integration of various services is a distributed process. This requires some form of distributed process management with language for process control and monitoring. The domain of this control and monitoring language is not the application itself but rather the process executing the application.

Distributed Workflow

You'll notice that we have not yet mentioned status messages. We need feedback on the RSN process's status and a way to fix any problems. Going back to workflow is one way to address this requirement; some systems use XML to express a distributed workflow, without needing a central process engine.[12,13] These particular systems are based on the WfMC standard, which won't be widely adopted and has no standard for implementation evaluation. But the "process" message types in these systems substantially increase the level of process monitoring and control.

There is much research to be done in this area, especially if we aren't constrained by the formats current vendors suggest. As a naive start, consider these definitions of a hypothetical distributed PXML with ECL primitives step and response:

- *Step* has a step name, the prior step's name (and the action that led to this step), the sender's role in this current step, the current action requested, one or more receivers of the next step, a reference to XML types in the message body to act on, a reference to any superprocess of which this step might be a subprocess, and informal text to include in any accompanying emails to the step's current recipient.
- *Response* defines the exception action to take if a step is not performed as expected by a certain time. It includes a list of people or programs to notify.

With the right semantics and a few simple XML tags, you could define (completely apart from the work to be done) descriptions of the process to be accomplished. Each actor could process the incoming process step information independently, or there could be a host engine. In the former case, each actor might be able to modify the step as desired and to use new services and providers at runtime. When a process step is not performed as defined, the response determines whom to notify. Certainly, technical issues must be worked out with this approach, but they probably aren't as difficult to achieve as acceptance of a new way of doing business.

A crucial research question, as yet unexamined in any of the systems discussed so far, is how to control the flow of status messages. This is crucial because different participants should have access only to certain information about status in order to understand context. The issue of transparency is that they should have enough information to reliably and efficiently provide their services, but not enough to compromise the proprietary interests of upstream consumers of these services. For instance, should each actor be able to see the step just prior to the one to be executed? DAML-S does not yet address the issues of status monitoring and control, although it does acknowledge them.

The ACL/ECL Coordination Approach

Maintaining the state of such a distributed process is also a difficult problem, but one for which known techniques exist even for distributed planning and execution of tasks and services.[8] Techniques exist in the software agent community for coordinating cooperative problem solving[14] and corresponding ACL primitives. Clearly, a Web service ECL could benefit from this prior work on multiagent systems (MASs).

An advanced but important functionality is service integration planning.[15,16] Suppose a US bank is managing a mortgage. The bank might have the goal of insuring the house for US$200,000, which may not be possible. However, a smart planning system could realize that it can decompose the original goal into two subgoals, insuring each for US$100,000. Compensating transactions are insufficient for such cases.

A set of AI planning technologies can address such problems, including replanning due to contingencies. Achieving explicit goals is a fundamental part of planning technology,[17] and services will need to advertise pre- and postconditions to use these techniques. This is particularly important when services have actions that affect which service is needed next, perhaps including undoing the action just taken.

Service Agents

So why not just use MAS technology? The academic approach hasn't solved the fundamental problems of service discovery yet. How do interested parties find the capability they're looking for? How do they advertise? How can software do this on behalf (but without the intervention) of a person? Many proposals along these lines exist, ranging from content-based routing in the early 1990s to this year's use of DAML-S for agents,[18] but none of them has led to deployed, practical systems. DAML-S itself does not solve the discovery problem because it doesn't provide a standard ontology of business transaction concepts, such as payment.

Although the MAS community is attempting to conform to emerging WSxL/XML standards, and excellent efforts in this direction have appeared on the horizon (see www.cs.georgetown.edu/~blakeb/AgentB2B/blake_AgentB2B_Position.pdf), we believe another approach might succeed more quickly. Specifically, Web services might evolve into software agents by salvaging academic software systems.

A software agent accepts any text message from anyone over the Internet and makes no commitment to the kind of response or any future message. Part of the WSxL approach's success has been to require providers to advertise services as rather fine-grained operations that accept and generate only certain messages (and at specific addresses), which are readable in advance of sending messages.

Suppose an ECL were implemented simply as WSDL operations. Each operation would describe its I/O messages via XML complex types and a schema. Any other service considering sending a `Create-order`, `Step`, or `Commit-Action` ECL message could read this by using the appropriate ECL operation for that service. The sender would then know what input message was appropriate and what output message to expect. This contrasts with the general ACL approach in which each agent is expected to parse whatever is sent.

In a previous work, one of us described a methodology in which a Web description described the messages each agent could send and receive.[19] This was for the benefit of human agent developers for coordination development. WSDL offers the possibility of machine-readable message descriptions that software can run at execution time, with some general standards and semantics evolving from the business world.

As Web services become more sophisticated, they effectively will become software agents themselves. We believe these new systems, or *service agents*, will use an ECL to coordinate distributed processes with no fixed process model, that they can be developed, and that they're necessary to bringing dynamic VEs forward. Academics will initially need to build unsatisfying technologies combining DAML-S with UDDI is a good example.[20] But academically uninteresting work is absolutely vital because there is no standard way to represent Web service authorization, and virtually no public Web services deliver real products, such as food or travel reservations.

To be clear, although lots of people are doing excellent academic research in Web services, we advocate that the academic software agent systems be discarded (except for research purposes or unless they stay hidden beneath WSDL technology). Industry developers should look at MAS technology and steal freely, and MAS researchers who want to make a difference should start from scratch and build on top of the emerging industrial Web service technologies. An academic might ask, "Why advocate throwing away good systems and developing on top of bad ones?" Our answer? Because this approach has the advantage of having never before been tried. Ignoring industrial technologies leads only to published papers, while ignoring well-studied advanced distributed computing principles can lead to slow industrial progress due to the necessity for re-invention based on experience. A first step is the collaboration between CommerceNet and the Stanford Center for Information Technology, which is producing workshops with industry on such topics (see www.commerce.net/events/ecoii-bsr-workshop.html). We will produce real business services that can leverage dormant academic technologies, ultimately resulting in the ability to program the world.

Acknowledgments

This work was done as part of the FX-Agents Project at Stanford University in partnership with NE, Intec Web, and Genome. The FX-Agents Project participants contributing to this work include Hans Bjornsson, Rada Chirkova, Jung Ung Min, Waqar Mohsin, Hidehito Gomi, Daishi Kato, Kyohei Kawazoe, Mike Kassoff, and Michael Genesereth. This work benefited greatly from outside contributions from Richard Jullig, Donald Steiner, Michael Kolb, and Marty Tenenbaum.

References

1. F. Curbera et al., "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, 2002, pp. 86–93.
2. S. Vinoski, "Web Services Interaction Models Part 1: Current Practice," *IEEE Internet Computing*, vol. 6, no. 3, 2002, pp. 89–81.
3. R. Trivedi, *The Role of Taxonomies in UDDI: tModels Demystified*, Junipermedia, 2002; www.developer.com/java/print.php/10922_1367781_2.
4. C. Petrie, "The XML Files," *IEEE Internet Computing*, vol. 2, no. 3, 1998; http://snrc.stanford.edu/~petrie/online/v2i3-webword.html.
5. B. Benatallah, Q. Sheng, and M. Dumas, "The Self-Serv Environment for Web Services Composition," *IEEE Internet Computing*, vol. 7, no. 1, 2003, pp. 40–48.
6. "Workflow Management Coalition Workflow Standard—Interoperability Wf-XML Binding," 2000; www.wfmc.org/standards/docs/Wf-XML-1.0.pdf.
7. J.F. Allen, J. Hendler, and A. Tate, eds., *Readings in Planning*, Morgan Kaufmann, 1990.
8. C. Petrie, S. Goldman, and A. Raquet, "Agent-Based Project Management," *LNAI*, vol. 1600, Springer-Verlag, 1999; www-cdr.Stanford.edu/ProcessLink/papers/DPM/dpm.html.
9. Y. Labrou, T. Finin, and Y. Peng, "The Current Landscape of Agent Communication Languages," *IEEE Intelligent Systems*, vol. 14, no. 2, 1999, pp. 45–52; http://umbc.edu/~finin/papers/ieee99.pdf.
10. R.G. Smith, "The Contract-Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *IEEE Trans. Computers*, vol. 29, no. 12, 1980, pp. 1104–1113.
11. D. Steiner and M. Kolb, "Enabling Business Process Connectivity," *WebV2*, 2002 ; http://snrc.stanford.edu/~petrie/

fx-agents/xserv/webv2.pdf.

12. M. zur Muehlen and F. Klein, "AFRICA: Workflow Interoperability Based on XML-Messages," *Proc. Workshop Infrastructures for Dynamic Business-to-Business Service Outsourcing* (ISDO'00), Information Soc. Development Office, 2000; www.wi.unimuenster.de/is/mitarbeiter/ismizu/MIXU.FLKL-AFRICA(CAiSE2000).pdf.
13. R. Tolksdorf, "Workspaces: A Web-Based Workflow Management System," *IEEE Internet Computing*, vol. 6, no. 5, 2002.
14. M. Singh, "Be Patient and Tolerate Imprecision: How Autonomous Agents Can Coordinate Effectively," *Proc. Int'l Joint Conf. Artificial Intelligence* (IJCAI), Morgan Kauffman, 1999, pp 512–517; www.csc.ncsu.edu/faculty/mpsingh/papers/mas/ijcai-99.ps.gz.
15. D. McDermott "Estimated-Regression Planning for Interactions with Web Services," *Proc. AI Planning Systems Conf.* (AIPS'02), AAAI Press, 2002; ftp://ftp.cs.yale.edu/pub/mcdermott/papers/aips02.pdf.
16. S. McIlraith and T. Son, "Adapting Golog for Composition of Semantic Web Services," *Proc. 8th Int'l Conf. Knowledge Representation and Reasoning* (KR2002), Morgan Kauffman, 2002; www.daml.org/services/mci-son-kr02.ps.
17. M. Papazoglou et al., *XSRL: An XML Web-Services Request Language*, tech. report #DIT-02-0079, Univ. Trento, Povo, Italy, 2002; www.ebpml.org/xsrl.zip.
18. T.R. Payne, R. Singh, and K. Sycara. "Calendar Agents on the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 3, 2002, pp. 84–86; http://dsonline.computer.org/0205/departments/sem.htm.
19. C. Petrie, "Agent-Based Software Engineering," *Agent-Oriented Software Eng.* (LNCS vol. 1957), P Ciancarini and M. Wooldridge, eds., Springer-Verlag, 2000, pp. 59–76; www.springer.de/cgi/svcat/search_book.pl?isbn=3-540-41594-7.
20. M. Paolucci et al., "Importing the Semantic Web in UDDI," to appear in *Proc. Web Services, E-Business and Semantic Web Workshop*, 2003; www.softwagents.ri.cmu.edu/papers/Essw.pdf.

Charles Petrie is a senior research scientist at Stanford University. His research interests include concurrent engineering and distributed process management. He received a PhD in computer science from the University of Texas at Austin. Contact him at petrie@stanford.edu.

Christoph Bussler is executive director at the Digital Enterprise Research Institute at the National University of Ireland in Galway. His research interests include B2B integration, process management, and Semantic Web services. He received a PhD in computer science from the University of Erlangen, Germany. He is also a member of the IEEE CS and the ACM. Contact him at ChBussler@aol.com; http://hometown.aol.com/ChBussler.