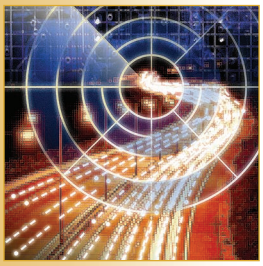

Editor: Michael N. Huhns • *huhns@sc.edu*

Multiagent Systems with Workflows

Industry and researchers have two different visions for the future of Web services. Industry wants to capitalize on Web service technology to automate business processes via centralized workflow enactment. Researchers are interested in the dynamic composition of Web services. The authors show how these two visions are points in a continuum and discuss a possible path for bridging the gap between them.

José M. Vidal
University of South Carolina

Paul Buhler
College of Charleston

Christian Stahl
Humboldt University, Berlin

The recent popularity of Web services and the Semantic Web has renewed people's interest in building large-scale, adaptive multiagent systems. Industry is largely interested in business process interoperation so it can automate processes such as purchasing and form processing.

The trend is to have workflow management systems enact statically defined compositions of Web services. The interaction among these services generates the workflow. However, researchers want to realize their long-term vision of dynamically composed Web services. As such, standardizing Web service transport and centralizing enactment mechanisms are important development steps toward future adaptive multiagent-based workflow systems.

In this article, we discuss a possible path for gradually bridging this gap between the centralized mindset on current Web service platforms and researchers' vision of distributed, dynamic Web service composition.

Static Workflow Specifications

Many businesses are interested in workflow automation. A workflow, for our purposes, is a series of actions performed by a series of actors. In a car insurance company, for example, a workflow instance starts whenever a driver submits a new claim. As part of the workflow, various experts step in at different stages: one might look at the wrecked car, another might check the driver's accident record, someone else might provide a repair estimate, and so on. Each expert contributes something to the claim, with the workflow ending when a final expert decides the total amount to award. Different agents can do many of the actions in the workflow in parallel, but some actions have temporal or conditional dependencies among them. It is the workflow description language's job to unambiguously declare all these dependencies.

The Business Process Execution Language for Web Services (BPEL4WS) is an XML-based language for describing workflows (see the sidebar).[1] With support from Microsoft and IBM, it's the current de facto standard for workflow description. A workflow described in BPEL4WS details the flow of control and any data depen-

 Published by the IEEE Computer Society 1089-7801/04/$20.00

The Business Process Execution Language for Web Services (BPEL4WS)

A BPEL4WS workflow description is a structured XML document; as such, a collection of tags defines the BPEL4WS language's vocabulary. Here's a summary of the primary tags and their meanings:

- <partners> contains a list of the Web services invoked as part of this workflow;
- <variables> contains the variables used in this workflow;
- <correlationSets> provides a way to specify precedences and correlations between Web service invocations that cannot be expressed as part of the main workflow;
- <faultHandlers> contains exception-handling routines;
- <compensationHandler> handles compensation actions if a transaction rollback occurs; and
- <eventHandlers> show how the workflow handles external (asynchronous) events.

Workflow logic is expressed with tags that map to traditional control flow structures:

- <sequence> executes the contents in a sequence,
- <flow> executes the contents in parallel,
- <while> implements a while loop,
- <switch> implements a case statement, and
- <pick> waits for external event then performs the activity associated with that event.

Within control flow structures, BPEL4WS defines tags that specify what activities to perform. These include the following:

- <invoke> invokes a specific Web service,
- <receive> receives an invocation message,
- <reply> sends a response message, and
- <assign> assigns a value, perhaps from a received message, to a variable.

The full BPEL4WS specification describes detailed semantics for the complete set of allowable tags. Additionally, the specification includes an XML Schema for BPEL4WS that can be used to validate syntactic correctness.

dencies among a collection of Web services being composed. When enacted, the composition itself becomes available as a meta-Web service, eligible for inclusion in other compositions. BPEL4WS requires that all Web services be described with available Web Services Definition Language (WSDL) descriptions.[2] The expectation is that domain experts will write workflow descriptions encoded in BPEL4WS, so these workflows won't change until the experts that wrote them decide to modify them.

Industry likes workflows because they offer predictable performance: once a company develops a workflow, the actors are expected to follow it to the letter. Moreover, the workflow's implementers can analyze and modify it if data from past experiences shows inefficiencies. These workflows also have some degree of fault tolerance thanks to their fault-handling mechanisms. Unfortunately, they remain largely rigid, which means they can't exploit resources for unexpected events.

Due to industry's increased focus on business process management (BPM) and acceptance of BPEL4WS, vendors are producing new software tools for workflow design, specification, and enactment. An example of one such tool is IBM's BPEL4WS Java Runtime (BPWS4J) platform. Think of the BPWS4J engine as an interpreter for the workflow specification: when the engine receives a workflow description, it enacts the workflow in a centralized manner. The engine manages each Web service's interaction in the workflow, ensuring that all operations are performed as specified in the BPEL4WS description. The downside to this approach is that, although the engine can execute these invocations asynchronously (thus generating some degree of parallelism), the process is still centralized, which means it suffers from the single point-of-failure weaknesses that plague centralized designs. This approach to workflow enactment should not, however, come as a surprise because it's exactly the type that BPEL4WS was designed to describe.

Dynamic Composition via DAML-S

At the other end of the flexibility spectrum, we have the DAML-based Web Service Ontology (DAML-S)[3] and dynamic Web service composition. This approach's supporters propose that every Web service be described with DAML-S's inputs, outputs, preconditions, and effects (IOPEs) so that when someone needs a specific service that no existing service can deliver, an AI planner can make it happen. If a user had $5,000, for example, and wanted to go on vacation, the planner would pull together a series of services to transform the money into goods (such as airplane tickets, hotel reservations, and so on). This plan is similar to a workflow because it describes a set of steps toward a goal, but its functionality is more limited than a

workflow's. Plans are simple sequences of actions whereas a workflow can contain `while` loops, `if-then` conditions, and so on.

Obviously, if we dynamically compose Web services into plans that achieve specified objectives, we get much more flexible solutions than static workflows. However, this approach is not without its drawbacks: as we just mentioned, AI planners generate only sequences of actions. To obtain a workflow's level of sophistication, we'll either have to develop more sophisticated planners or perform frequent replanning. Unfortunately, due to the planning process's computational complexity, planners do not scale well as the number of available services increases. Dynamic composition becomes feasible only when already-available Web services describe themselves with DAML-S's IOPEs and use the same ontologies.

Multiagent Workflow Enactment

Decentralized, multiagent workflow-enactment techniques can bridge the gap between static workflow enactment and dynamic Web service composition. Static workflows fall on the rigid, but computationally cheap, end of the spectrum, whereas dynamic composition falls at the opposite end: flexible but computationally expensive. Multiagent workflow enactment falls somewhere in the middle, and it has many implementation options, each of which lands at a different point in the spectrum.

Initially, we'll constrain the discussion to the simplest case in which we eliminate the centralized workflow engine, and a decentralized collection of cooperating agents assumes responsibility for maintaining the workflow's integrity. Notably, this simple scenario is complicated by the fact that BPEL4WS was not designed for multiagent enactment and, therefore, lacks explicit instructions about how agents should coordinate. Other workflow description languages are more amenable to multiagent enactment,[4] but by using BPEL4WS, we can exploit existing workflows and tools. As such, we must first develop methods for performing the BPEL4WS-to-multiagent-enactment mapping.

Functional Equivalency

Our first task when mapping from BPEL4WS to a multiagent enactment is to make sure that the new enactment is functionally equivalent to the centralized version. As part of this task, we also must decide how to allocate services among agents. If enough agents are available, we simply give each service to a different agent, but even in a simple scenario like this one, it's not clear if such an allocation is optimal. For example, if the output from one service is a large picture that another service must receive for processing, then it might be better to give these two services to one agent to minimize the amount of communication. But how do we determine where an agent should next forward its results, or if we've attained the proper workflow?

The simple answer is to give each agent explicit directions about what to do once it receives a service invocation. For example, if services A and B must run in sequence, then the agent responsible for A must invoke B right after it finishes its invocation. More generally, we can transform a workflow into different Petri nets, depending on the chosen mapping between services and agents.[5,6] Petri nets combine a precise mathematical formalism with an intuitive graphical representation. A Petri net $N = (P, T, F)$ consists of a set of *transitions* T (boxes), a set of *places* P (ellipses), and a *flow relation* F (arcs).[7] A transition represents an active element, and a place is a passive element. Petri nets are well suited for modeling workflow processes.[8] We can then test the Petri nets using one of many available simulation tools to determine if the resulting workflow is functionally equivalent, if any bottlenecks exist, and so on.

From BPEL4WS to Petri Nets

We build every process in a BPEL4WS workflow by plugging language constructs together; we thus can translate each construct of the language into a Petri net. Such a net forms a pattern of the respective construct, and each pattern has an interface for joining it with other patterns, as is done with BPEL4WS constructs. Some of the patterns have parameters — for example, some constructs have inner constructs. The respective pattern must be able to carry any number of inner constructs, as its equivalent in BPEL4WS can do. We aim to keep all constructs' properties in the patterns. The collection of patterns generates a Petri net semantics for BPEL4WS.

Figure 1 shows the pattern for the BPEL4WS's `receive` construct. This construct receives a message that it saves in a variable unless a fault is thrown because of a mangled message or some other error. In general, a dashed box frames a pattern, and inside this frame, the net models the corresponding BPEL4WS construct's structure. The nodes depicted directly on the frame establish the interface. Control flows from top to bottom; communication between processes flows horizontally.

Outside the frame, external objects relate to the scope. Petri net patterns for each BPEL4WS construct already exist, so by plugging the patterns together and nesting them, we can translate every workflow specified in BPEL4WS into a Petri net.

Once we have a Petri net representation of the workflow, we can analyze it by testing it on a simulator. Specifying the Petri net with a broadly accepted standard format increases the set of tools we can use for analysis. For this reason, we use the XML-based Petri Net Markup Language.[9] PNML also supports a module concept that lets modules reference one another using their well-defined interfaces.[10]

Figure 2 summarizes the transformation from a BPEL4WS description to a PNML file. The BPEL4WS workflow process `proc.bpel` is the input to the parser, which is a collection of EXtensible Stylesheet Language Transformations (XSLT)[11] templates along with the PNML modules. The parser replaces each construct in the workflow with its corresponding PNML module and glues their interfaces together. The end result is a Petri net called `proc.pnml` in PNML.

Beyond Functional Equivalence

As we move beyond ensuring simple functional equivalence, the agents' autonomous nature demonstrates its importance. The system becomes more flexible as the agents gain more autonomy in their decision-making; simultaneously, these techniques extract a cost in terms of runtime computational complexity. As such, we can place the various systems that we'll discuss in this section in a spectrum (see Figure 3, next page). Specifically, we'll consider a slightly more sophisticated scenario than the one we just used. We start by replacing one atomic service with another, then one service with a group of services, and finally, we replace one group with another group. We can base these replacements on straightforward matching or on more complex similarity measures.

Moving further, agents learn from the services they represented in the past to make better future matches. Ultimately, we can consider a community of selfish agents in which many workflow instances are enacted, although the agents might disagree about which workflow instances exist at any point in time. The agents in this environment make decisions about which actions to take based on their beliefs about the currently instantiated workflows and their expected payoffs.

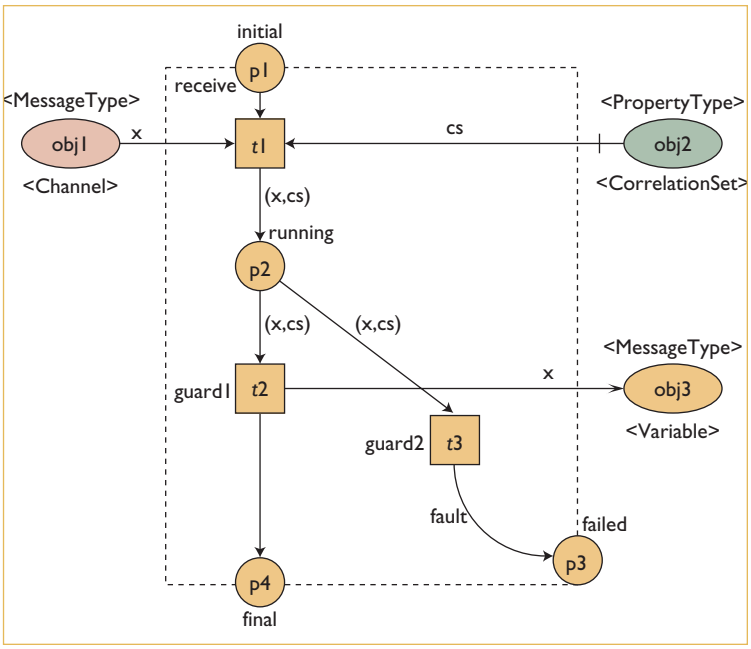

Figure 1. Petri net pattern for the BPEL4WS `receive` construct. When the pattern is activated, it's executed in two steps. First, the message is taken from the `channel`, and the `CorrelationSet` is read (`t1`). In the second step, this information is analyzed. Either a fault occurs (`t3`) or the message is written into the variable (`t2`). In both cases, the pattern is finished.

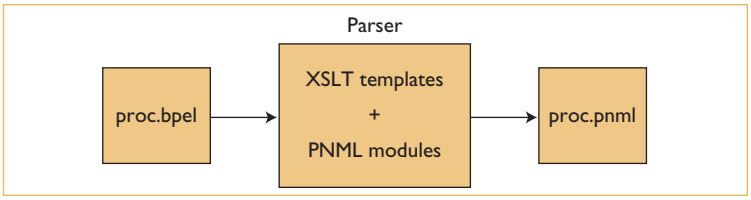

Figure 2. Transformation of BPEL4WS to the Petri Net Markup Language (PNML). A BPEL4WS workflow process `proc.bpel` is the input to the parser. Using XSLT templates and the PNML modules, the process is translated into a process `proc.pnml` in PNML format.

Enactment with Substitutions

BPEL4WS uses WSDL to identify the Web services to be invoked. Because WSDL does not provide any semantics on a service's effects, it's unsuitable as a tool for finding interchangeable services. The most common solution is to use DAML-S instead, meaning we change the workflow to use the DAML-S descriptions of the needed services instead of using WSDL. This change is akin to having the workflow description describe the services' effects rather than providing the services' names.

Once we make this change, we can use a service's semantic description to find a replacement

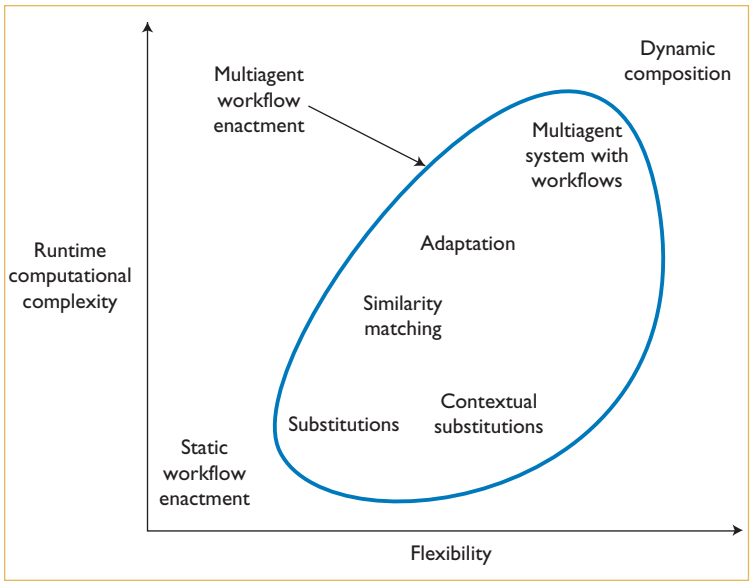

Figure 3. The spectrum of possible Web service composition scenarios. The x-axis ranges from static workflows completely pre-determined at the outset to very flexible systems that generate new workflows and plans as needed. The y-axis ranges from minimal runtime complexity to systems that require very large computational resources at runtime. The various multiagent workflow enactment techniques occupy the center of this space.

service that is either an exact match or more general than the one currently specified.[12] We could also replace it with a more specific service than the current one, but we must be careful because it might break the workflow. Replacing a current-weather service with a current-temperature service, for example, could lead to problems later if we need to know the wind speed.

Replacing one service with another that provides an exact or more general service is easy, but replacing a service with a group of services requires a plan. This type of substitution pushes us further toward the dynamic end of the spectrum. In fact, the search for a suitable replacement set might require searching over all existing services and thus increase complexity. We can limit the search by considering only a subset of the available services.

Similarity Matching

Matching algorithms try to determine if two services are similar enough to substitute for each other. This matching goes beyond simple subsumption matching; it uses domain knowledge about the particular services' ontologies. Services' descriptions lie within their IOPEs, which can use any ontology, so we need domain knowledge about these ontologies to refine our search. For example, if a service provides a weather forecast (from the weather ontology) as an output, we need to know exactly how similar this weather forecast is to a severe weather warning, a temperature forecast, a wind advisory, and so on.

Because similarity matching requires intimate knowledge of the domain ontology, it seems unlikely that we could develop domain-independent algorithms for it. At best, we might hope to develop algorithms that take as input the particular ontologies (along with specific domain knowledge about them) and match from there. We then could express this specific domain knowledge in a standardized ontology that could describe the similarity or replaceability of concepts within the Web service's context. It's hard to imagine how this ontology might look without the benefit of many sample cases, but we expect it would represent concepts such as the composition of an item (even if a service for it doesn't exist yet), or two items being similar to each other when we're concerned only with a particular service.

Notice that the problem of matching services is similar, but not identical, to the problem of matching ontologies. Matching ontologies requires matching terms from one ontology to another in order to determine, for example, that what one ontology calls "automobile" is the same thing another one calls "car." When matching services, we assume that this ontology matching has already occurred and that the agent knows all possible equivalencies between terms. The problem then becomes finding matches between different terms that, within the appropriate service's context, can be suitable substitutes for each other.

Contextual Substitutions

We can find more, perhaps better, matches by analyzing the service's place within a workflow. If only one of a service's several outputs is used later in the workflow, for example, then it's probably safe to replace that service with another that provides only that one output. BPEL4WS's structure enables this because it shows how results from services are stored in variables or databases; we access these results later in the workflow as inputs to other services. More complex contextual substitutions look at the `effects` part of the service description to determine which effects serve as preconditions for other services. In general, this process strives to determine the minimal set of IOPEs necessary to replace a given service. As such, it assumes that the workflow's designer has

failed to specify the service's minimal requirements — a reasonable assumption because new services likely have been added since the designer wrote the workflow. Think of this process as a form of workflow optimization performed when the designer creates or updates the workflow, or when new service descriptions arise.

Adaptation

Certain aspects of the service might not be captured or might be incorrectly specified in the ontologies that describe it. When an agent consumes a service and detects these anomalies, it can choose to update its own model of the service to better reflect reality. For example, a stock-quote service might deliver stock quotes consistently 30 seconds later than another identical (according to the IOPE description) service. The service's consumer might want to remember this fact and use the faster service next time.

These techniques build on past agent-modeling research[13,14] by integrating it with the use of ontologies. Agent-modeling research is based on utility functions and the assumption that all we can know about an agent must be learned via interactions — by applying the utility function to the services the agent renders. If we also have a description of both the desired and provided services, we can use these descriptions to influence the utility function, perhaps modifying its results. In this way, agents could build accurate models of other agents with fewer interactions. In machine-learning terms, this technique amounts to using the ontology match to reduce the set of possible hypotheses (descriptions) and then using the interactions (sample data) to pick the best hypothesis from those remaining.

Once an agent has built models of other agents, we expect it to want to share them. Think of these models as filters or modifiers of the published descriptions. The new descriptions contain semantics and utility, unlike many recommender systems, which deal only with utilities. One agent should be able to explain to another why a particular service did not meet its criteria. For example, it could state, "although the service claims to be speedy, my experience is that sometimes it takes too long, but it does return the data as advertised." To dynamically adapt a workflow, we can thus combine the power of an ontology of services with existing techniques from recommender systems.

Multiagent Systems with Workflows

Looking further into the future, we foresee systems veering increasingly toward the adaptive end of the spectrum. In these workflow-based multiagent systems, the service providers are agents themselves; they therefore acquire the full proactive, autonomous, and selfish characteristics that we normally associate with agency. In these systems, it's no longer clear which workflow instances are currently active. For example, a company that does contract-based consulting has many workflows that describe how to go from an idea or a call for proposals to a contract award. The individuals in this company receive a reward proportional to the number of awards they help realize. As such, they take actions that maximize the expected number of workflows achieved. However, individuals on a team might disagree about which potential contracts the company is currently developing.

In this scenario, we abandon the workflow-based view and acquire a multiagent-based approach. The agents take actions that push possible workflow instances further without knowing which instances currently exist. They also must sometimes make complex decisions about which actions to take to maximize the expected utility. For example, a simple decision arises when the agent receives a complete proposal and is asked to submit it to the appropriate client. Because this action is near the end of the workflow, the agent can easily determine that taking the action will lead to a successful workflow termination. On the other hand, an agent might be asked to write a summary in response to a client's call for proposals. In this case, it's so early in the process that it's hard for the agent to determine the likelihood that the action will lead to the workflow's completion. Therefore, we need some rules or methods that help the agents determine a given proposal's expected likelihood of success. These rules will require some knowledge of current and past system dynamics: who is doing what, who has done what in the past, and so on.

Workflows are used in this case as blueprints for orchestrating the system's dynamics. Although the agents are free to diverge somewhat from the available workflows, they are not free to assemble entirely new workflows. As such, the designer maintains some control over the system's dynamics while still letting the agents exploit any opportunities that might arise. Workflows are also less computationally expensive than the planning that must occur if we let agents dynamically compose their own plans.

Unlike traditional multiagent systems based on

joint plans and intentions, which require agents first to jointly commit to a plan and then to execute it, the agents in our workflow-based system are completely opportunistic and never commit to finishing a workflow. Flexibility means that the designer must structure the payoffs in some way that creates the proper incentives for the agents to finish the workflow. An advantage of this flexibility is that no synchronization bottlenecks crop up in which the agents needed to fulfill a particular workflow must all agree to participate. The workflow can get started even before all the required agents are available.

Moving Forward

The path we describe in this article will let us quickly produce systems that meet the current, often desperate, need for interoperation among companies while still building systems that developers can further modify to include dynamic adaptation and composition. We are currently developing tools for mapping BPEL4WS workflows into Petri nets that we can use to generate multiagent instantiations of the workflow. We will then be able to run tests on these Petri nets to determine various instantiation algorithms' benefits and drawbacks. After this, the next stage is the development of algorithms for effective similarity matching and contextual substitutions. These algorithms would let agents intelligently diverge from prescribed workflows when needed. We believe that these algorithms will require induction and machine-learning techniques along with deduction methods.

Acknowledgments

The US National Science Foundation funded part of this work under grant 0092593.

References

1. T. Andrews et al., *Business Process Execution Language for Web Services*, IBM, version 1.1, 2nd public draft release, May 2003; www.ibm.com/developerworks/webservices/library/ws-bpel.
2. E. Christensen et al., *Web Services Description Language*, version 1.1, World Wide Web Consortium recommendation, Mar. 2001; www.w3.org/TR/wsdl.
3. D.Martin et al., *DAML-S Specification*, version 0.9, 2003; www.daml.org/services/daml-s/.
4. M.P. Singh, "Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition," *Proc. 2nd Int'l Joint Conf. Autonomous Agents and Multiagent Systems*, ACM Press, 2003, pp. 907–914.
5. P. Buhler and J.M. Vidal, "Semantic Web Services as Agent Behaviors," *Agentcities: Challenges in Open Agent Environments*, B. Burg et al., eds., Springer-Verlag, 2003, pp. 25–31.
6. P. Buhler, J.M. Vidal, and H. Verhagen, "Adaptive Workflow = Web Services + Agents," *Proc. Int'l Conf. Web Services*, CSREA Press, 2003, pp. 131–137.
7. W. Reisig, *Petri Nets*, Springer-Verlag, 1985.
8. W.M.P. van der Aalst, "The Application of Petri Nets to Workflow Management," *J. Circuits, Systems, and Computers*, vol. 8, no. 1, 1998, pp. 21–66.
9. J. Billington, et al., "The Petri Net Markup Language: Concepts, Technology, and Tools," *Proc. Int'l Conf. Applications and Theory of Petri Nets 2003*, W. van der Aalst and E. Best eds., LNCS, vol. 2679, Springer, 2003, pp. 483–505.
10. E. Kindler and M. Weber, "A Universal Module Concept for Petri Nets. An Implementation Oriented Approach," *Proc. Workshop Algorithmen und Werkzeuge für Petrinetze AWPN 2001*, G. Juhás and R. Lorenz, eds., Fachberichte Informatik Katholische Universität Eichstätt, 2001, pp. 7–12.
11. J. Clark, *XSL Transformations (XSLT)*, ver. 1.0, W3C recommendation, Nov. 1999; www.w3.org/TR/xslt.
12. M. Paolucci et al., "Semantic Matching of Web Services Capabilities," *Proc. 1st Int'l Semantic Web Conf.*, LNCS, 2342, Springer, 2002, pp. 333–347.
13. P.J. Gmytrasiewicz and E.H. Durfee, "A Rigorous, Operational Formalization of Recursive Modeling," *Proc. 1st Int'l Conf. Multi-Agent Systems*, AAAI/MIT Press, 1995, pp. 125–132.
14. J.M. Vidal and E.H. Durfee, "Learning Nested Models in an Information Economy," *J. Experimental and Theoretical Artificial Intelligence*, vol. 10, no. 3, 1998, pp. 291–308.

José M. Vidal is an assistant professor in the Department of Computer Science and Engineering at the University of South Carolina. His technical interests include multiagent systems and distributed programming. He received a PhD from the University of Michigan in computer science and engineering. He is a member of the IEEE, the ACM, and AAAI. Contact him at vidal@sc.edu.

Paul Buhler is an assistant professor in the Department of Computer Science at the College of Charleston. He also is a PhD candidate in computer engineering at the University of South Carolina. His research interests encompass compositional strategies for loosely coupled, distributed software systems. He holds an MS in computer science from Johns Hopkins University. He is a member of the IEEE Computer Society and the ACM. Contact him at pbuhler@cs.cofc.edu.

Christian Stahl is a computer science student at Humboldt University, Berlin. His technical interests include business process modeling and Petri nets. He is a member of a research group that focuses on developing formal methods for specification, modeling, and verification of distributed systems, particularly business processes. Contact him at stahl@informatik.hu-berlin.de.