Editor: Siobhán Clarke • siobhan.clarke@cs.tcd.ie

Current Solutions for Web Service Composition

Nikola Milanovic and Miroslaw Malek • *Humboldt University, Berlin*

Web service composition lets developers create applications on top of service-oriented computing's native description, discovery, and communication capabilities. Such applications are rapidly deployable and offer developers reuse possibilities and users seamless access to a variety of complex services. There are many existing approaches to service composition, ranging from abstract methods to those aiming to be industry standards. The authors describe four key issues for Web service composition.

In service-oriented computing (SOC), developers use services as fundamental elements in their application-development processes. Services are platform- and network-independent operations that clients or other services invoke. To operate in an SOC environment, services must overtly define their properties in a standard, machine-readable format. SOC thus offers three native capabilities: description, discovery, and communication.[1] Web services are a typical SOC example: developers implement SOC native capabilities using Web Services Description Language (for description), Universal Description, Discovery, and Integration (for discovery), and SOAP (for communication).[2]

To create applications, SOC developers use service composition, which they introduce on top of SOC's native capabilities. Developers and users can then solve complex problems by combining available basic services and ordering them to best suit their problem requirements. Service composition accelerates rapid application development, service reuse, and complex service consummation. Currently, however, service composition isn't standardized, nor does it include definitions of the key requirements that every composition approach must satisfy (such as scalability, dependability, and correctness). If the SOC paradigm is to succeed and become the dominant architecture of future distributed systems, we must provide a stable and dependable service composition solution.

Here, we offer a survey of existing proposals for Web service composition, and compare them with respect to four key requirements, which we discuss in the next section. By offering this overview and systematization of key properties, as well as a constructive critique of existing approaches, we hope to help service-composition designers and developers focus their efforts and deliver more usable and durable solutions, while also addressing the technology's critical needs.

Service Composition Requirements

The complexities of distributed systems and increasing trust barriers have influenced SOC evolution at the hardware, operating system, and application layers. Although modern operating systems can also be seen as sets of collaborating services, in this survey, we focus on the application layer. From the developer's perspective, service composition offers reuse possibilities. From the user's perspective, it offers seamless access to a variety of complex services.

Service composition requirements differ from those of mainstream component-based software development. In place of access to documentation or code (either source or binary), SOC application developers and users have access only to WSDL's rudimentary functional descriptions. Services execute in different containers, separated by firewalls and other trust barriers. A composition mechanism

must therefore satisfy several requirements: connectivity, nonfunctional quality-of-service properties, correctness, and scalability.

Every composition approach must guarantee *connectivity*. With reliable connectivity, we can determine which services are composed and reason about the input and output messages. Because Web services are based on message passing, however, developers must also address *nonfunctional QoS properties*, such as timeliness, security, and dependability. Composition *correctness* requires verification of the composed service's properties, such as security or dependability. Finally, because complicated business transactions are likely to involve multiple services in a complex invocation chain, composition frameworks must *scale* with the number of composed services.

We can illustrate the need for such requirements with two examples. First, suppose we have a trusted and an untrusted service, where the service architecture defines trust. What happens when we compose these services in sequence? Is this composition trusted, untrusted, or something in between? It's crucial that we know whether our application is secure and dependable. And what happens when we compose two trusted services? Do we assume that the composition of trusted services will also be trusted?

Another example that demonstrates the need for nonfunctional properties is a composition's timeliness. Suppose we have a simple handshaking example with two partner services, in which one wants to invoke a method on the other. The client service expects to be notified when it can apply (invoke a method), while the provider service expects to be notified that the client wants to utilize its service. Unless its developers understand such requirements in advance, the composition will not produce useful or expected results.

Web Service Composition Approaches

Once Web services' native capabilities were fully developed, service composition approaches began emerging. Because the first-generation composition languages — IBM's Web Service Flow Language (WSFL) and BEA Systems' Web Services Choreography Interface (WSCI) — were incompatible, researchers developed second-generation languages, such as the Business Process Execution Language for Web Services (BPEL4WS, or BPEL for short), which combines WSFL and WSCI with Microsoft's XLANG specification. Nonetheless, the Web Services Architecture Stack still lacks a process-layer standard for aggregation, choreography, and composition (www.w3.org/2002/ws). Here, we examine several of the proposals for Web services composition, comparing how they meet requirements for connectivity, nonfunctional properties, correctness, and scalability.

BPEL

BPEL (www.ibm.com/developerworks/library/ws-bpel) is an XML language that supports process-oriented service composition.[3] Developed by BEA, IBM, Microsoft, SAP, and Siebel, BPEL is currently being standardized by the Organization for the Advancement of Structured Information Standards (www.oasis-open.org). (Sun Microsystems recently joined the OASIS technical committee as well.)

BPEL composition interacts with a Web services' subset to achieve a given a task. In BPEL, the composition result is called a *process*, participating services are *partners*, and message exchange or intermediate result transformation is called an *activity*. A process thus consists of a set of activities. A process interacts with external partner services through a WSDL interface.

To define a process, we use

- a BPEL source file (.bpel), which describes activities;
- a process interface (.wsdl), which describes ports of a composed service; and
- an optional deployment descriptor (.xml), which contains the partner services' physical locations (a partner service's implementation and location can be changed without modifying the source file).

BPEL has several element groups, but the basic ones are

- process initiation: `<process>`
- definition of services participating in composition: `<partnerLink>`
- synchronous and asynchronous calls: `<invoke>`, `<invoke>...` `<receive>`
- intermediate variables and results manipulation: `<variable>`, `<assign>`, `<copy>`
- error handling: `<scope>`, `<faultHandlers>`
- sequential and parallel execution: `<sequence>`, `<flow>`
- logic control: `<switch>`

As an example, we'll model the composition of

three services. Service *A* is called synchronously and starts a process. Two asynchronous services, *B* and *C*, are then called in parallel using Service *A*'s output as their input. The process waits for their completion and then makes a decision based on the results. The stripped BPEL code for this composition follows (for clarity, we've omitted much of the code and assumed that all services offer only one operation at one port):

```
<process name="test">
  <partnerLinks>
    <partnerLink name="client"/>
    <partnerLink name="serviceA"/>
    <partnerLink name="serviceB"/>
    <partnerLink name="serviceC"/>
  </partnerLinks>
  <variables>
    <variable name="processInput"/>
    <variable name="AInput"/>
    <variable name="AOutput"/>
    <variable name="BCInput"/>
    <variable name="BOutput"/>
    <variable name="COutput"/>
    <variable name="processOutput"/>
    <variable name="AError"/>
  </variables>
  <sequence>
    <receive name="receiveInput" variable="input"/>
    <assign><copy>
      <from variable="processInput"/>
      <to variable="AInput"/>
    </copy></assign>
    <scope>
      <faultHandlers>
        <catch faultName="faultA" faultVariable="AError"/>
      </faultHandlers>
      <sequence>
        <invoke name="invokeA" partnerLink="serviceA"
          inputVariable="AInput" outputVariable="AOutput"/>
      </sequence>
    </scope>
    <assign><copy>
      <from variable="AOutput"/>
      <to variable="BCInput"/>
    </copy></assign>
    <flow>
      <sequence>
        <invoke name="invokeB" partnerLink="serviceB"
          inputVariable="BCInput"/>
        <receive name="receive_invokeB" partnerLink="serviceB"
          variable="BOutput"/>
      </sequence>
      <sequence>
        <invoke name="invokeC" partnerLink="serviceC"
          inputVariable="BCInput"/>
        <receive name="receive_invokeC" partnerLink="serviceC"
          variable="COutput"/>
      </sequence>
    </flow>
    <switch><case>
      <!– assign value to processOutput –>
    </case></switch>
    <invoke name="reply" partnerLink="client"
      inputVariable="processOutput"/>
  </sequence>
</process>
```

Researchers recently released BPELJ (www-106.ibm.com/developerworks/webservices/library/ws-bpelj/), a combination of BPEL and Java that lets developers include Java code inside BPEL code. Developers can thus use Java "snippets" to perform intermediate transformations such as value calculations within documents; document construction and deconstruction using information from other documents and variables; and value calculations for flow controls. They can also perform side-effects without creating separate Web services. A snippet can assume it's running inside a J2EE container. It has access to all variables and partner links that are in its location's scope. We can use a snippet, for example, to write the `<switch>` construct omitted from the previous example:

```
<bpelj:snippet>
  <bpelj:code>
    if (OutputB > OutputC)
      processOutput = outputB;
    else
      processOutput = outputC;
  </bpelj:code>
</bpelj:snippet>
```

Developers can use BPEL with two other specifications:

- Web Services-Coordination (www-106.ibm.com/developerworks/library/ws-coor/) coordinates Web services' actions when a consistent agreement must be reached on the service activities' outcome.
- Web Services-Transaction (www-106.ibm.com/developerworks/library/ws-transpec/) defines Web services' transactional behavior.

There are several BPEL orchestration server implementations for both J2EE and .NET platforms, including IBM WebSphere (www-306.ibm.com/software/info1/websphere), Oracle BPEL Process Manager (formerly Collaxa BPEL Server; see www.oracle.com/technology/products/ias/bpel/), Microsoft BizTalk 2004 (www.microsoft.com/biztalk), OpenStorm ChoreoServer (www.openstorm.com), and Active BPEL (www.activebpel.org).

Semantic Web (OWL-S)

The Semantic Web vision is to make Web resources accessible by content as well as by keywords. Web services play an important role in this: users and software agents should be able to discover, compose, and invoke content using complex services. The DARPA Agent Markup Language (DAML) extends XML and the Resource Description Framework (RDF) to provide a set of constructs for creating machine-readable ontologies and markup information. The DAML program's Semantic Web contribution is the Web Ontology Language for Services (www.daml.org/services). OWL-S (previously known as DAML-S) is a services ontology that enables automatic service discovery, invocation, composition, interoperation, and execution monitoring.[4]

OWL-S models services using a three-part ontology:

- a service *profile* describes what the service requires from users and what it gives them;
- a service *model* specifies how the service works; and
- a service *grounding* gives information on how to use the service.

The process model is a service model subclass that describes a service in terms of inputs, outputs, preconditions, postconditions, and — if necessary — its own subprocesses. In the process model, we can describe composite processes and their dependencies and interactions. OWL-S distinguishes three types of processes: atomic, which have no subprocesses; simple, which are not directly invocable and are used as an abstraction element for either atomic or composite processes; and composite, which consist of subprocesses. Constituent processes are specified using flow-control constructs: `sequence`, `split`, `split+join`, `unordered`, `choice`, `if-then-else`, `iterate`, and `repeat-until`.

OWL-S would orchestrate the previous section's example as follows (again, using only the most important commands):

```
<daml:Class rdf:ID="test">
 <daml:subClassOf
rdf:resource="Process.CompositeProcess"/>
 <daml:subClassOf>
  <daml:Restriction>
   <daml:onProperty
rdf:resource="Process#composedOf"/>
   <daml:toClass>
    <daml:Class>
     <daml:intersectionOf rdf:parse-
Type="daml:collection">
      <daml:Class
rdf:about="process:Sequence">
       <daml:Restriction>
        <daml:onProperty
rdf:resource="Process#components"/>
        <daml:toClass>
         <daml:Class>
          <process:listOfInstancesOf
               rdf:parseType="daml:col-
lection">
           <daml:Class rdf:about="#ser-
viceA"/>
           <daml:Class
rdf:about="process:Split">
            <daml:Restriction>
             <daml:onProperty
rdf:resource="Process#components"/>
             <daml:toClass>
              <daml:Class>
               <process:listOfInstancesOf
                  rdf:parseType="daml:col-
lection">
                <daml:Class
rdf:about="#serviceB"/>
                <daml:Class
rdf:about="#serviceC"/>
               </process:listOfInstance-
sOf>
              </daml:Class>
        . . .
```

Researchers have proposed methods for transferring OWL-S descriptions to Prolog[5] and Petri-net-based notation[6] to further analyze verification. In the Prolog approach, the developer manually translates an OWL-S description to Prolog, which makes it possible to find an adequate plan for composing Web services for a target description. That is, for a given pool of available Web services, it's possible to use logical inference rules to automate service allocation for the required task. In the Petri-net approach, an OWL-S description is automatically translated into Petri nets. Developers use this representation to automate tasks such as simulation, validation, verification, composition, and performance analysis.

Web Components

The Web component approach treats services as components in order to support basic software-development principles such as reuse, specialization, and extension.[7] The main idea is to encapsulate composite-logic information inside a class definition, which represents a Web component. A Web component's public interface can then be published and used for discovery and reuse.

Composition logic comprises composition type and message dependency. Composition type can take two forms:

- *Order* determines whether a component can execute constituent services sequentially or in parallel.
- *Alternative execution* indicates whether a component can invoke alternative services until one succeeds.

Message dependency defines input and output message mapping. There are three types of dependency:

- *Synthesis* generates a composed service's output message by combining the output messages of constituent services.
- *Decomposition* binds the composed service's input messages into the input messages of constituent services.
- *Message mapping* allows custom mapping between constituent services' inputs and outputs.

The Web component approach supports several basic composition constructs: sequential, sequential alternative, parallel with result synchronization, and parallel alternative. They are augmented with `condition` and `while-do` constructs. A Web component class definition for our example is

```
class BC is paraWithSyn{
  public Msg BCInput, BCOutput;
  public operation(Msg)->Msg;
  private void compose(B.operation,
C.operation);
  private void
messageDecomposition(BCInput, BInput,
CInput);
  private void messageSynthesis(BOutput,
COutput, BCOutput);
}

class test us sequ {
  public Msg processInput, processOut-
put;
  public operation(Msg)->Msg;
  private void compose(A.operation,
BC.operation);
  private void messageDecomposition(pro-
cessInput, AInput);
  private void messageSynthesis(proces-
sOutput, BCOutput);
  private void messageMapping(AOutput,
BCInput);
}
```

We can specify a Web component in two isomorphic forms: a class definition and an XML specification described in Service Composition Specification Language. The SCSL specification consists of the composite service's interface and the composition logic. Composition logic is specified as follows in SCSL for the class `test` (defined above):

```
<construct>
 <composition type="sequ">
  <activity name="A">
   <input message="AInput"/>
   <output message="AOutput"/>
   <performedBy serviceProvider="A"/>
  </activity>
  <activity name="BC">
   <input message="BCInput"/>
   <output message="BCOutput"/>
   <performedBy serviceProvider="BC"/>
  </activity>
  <messageHandling>
   <messageDecomposition>
```

```
    <source message="processInput"/>
    <target message="AInput"/>
   </messageDecomposition>
   <messageSynthesis>
    <source message="BCOutput"/>
    <target message="processOutput"/>
   </messageSynthesis>
   <messageMapping>
    <source message="AOutput"/>
    <target message="BCInput"/>
   </messageMapping>
  </messageHandling>
 </composition>
</construct>
```

Web components offer both compatibility and conformance checking. Two services, $S1$ and $S2$, are compatible when $S1$ is at least as capable as $S2$, and when $S1$ can substitute for $S2$. Service $S1$ conforms to service $S2$ when we can combine $S1$ and $S2$ so that $S1$'s output can be taken as $S2$'s input. In our example, service A conforms to B and C, whereas B and C are compatible.

Algebraic Process Composition

Algebraic service composition aims to introduce much simpler descriptions than other approaches, and to model services as mobile processes to ensure verification of properties such as safety, liveness (correct termination, for example), and resource management.

Mobile-processes theory is based on π–calculus,[8] in which the basic entity is a process — it can be an empty process; a choice between several I/O operations and their continuations; a parallel composition; a recursive definition; or a recursive invocation. I/O operations can be input (receive) or output (send). For example, $x(y)$ denotes receiving tuple y on channel x; $\overline{x}[y]$ denotes sending tuple y on channel x. Dotted notation specifies an action sequence, such as $\overline{c}[1,d].d(x,y,z).\overline{c}[x+y+z]$, in which a process sends tuple $[1,d]$ on channel c, then receives a tuple at channel d whose components are bound to the variables x, y, and z, and finally sends the sum of $x+y+z$ to channel c. Parallel process composition is denoted with $A|B$. Several processes can execute in parallel and communicate using compatible channels.

Describing services in such an abstract way lets us reason about the composition's correctness.[9] Using π–calculus, we can describe our example composition as

$$A(processInput).\overline{B}[AOutput].\overline{C}[AOutput]|$$
$$B(BInput).\overline{out}[BOutput]|$$
$$C(CInput).\overline{out}[COutput]|out(processOutput)$$

Using simple reduction, we can see that the composition's only possible outcomes are either `processOutput=BOutput` or `processOutput=COutput`, which means that this composition guarantees lock freedom. In a finite number of steps, the composition will produce the desired result.

Apart from verifying liveness, we can treat other relevant properties by assigning behavioral types to processes. There are at least two possible ways to type processes: we can type only port subsets or type the entire process. In the first case, we can proscribe the type or shape of data that can be exchanged via two ports. In our example, this would create additional message limitations. We could, for example, require that both `AOutput` and `BInput` follow some pattern (type) to make reduction $\overline{B}[AOutput]|B(BInput)$ possible. In our current example, processes A and B can exchange any kind of message, but if we type the messages (ports), we could limit the exchange. In the second case, we type the entire process and the type notion becomes a homomorphic image of the process. In many such systems, process and type are synonyms.

With algebraic process composition, the general question is what information to type. Typing too little can make it impossible to verify some properties, such as security. On the other hand, typing too much creates a complexity that renders verification unusable or impractical.

Petri Nets

Petri nets are a well-established process-modeling approach. A Petri net is a directed, connected, and bipartite graph in which nodes represent places and transitions, and tokens occupy places. When there is at least one token in every place connected to a transition, that transition is enabled. An enabled transition might fire by removing one token from every input place, and depositing one token in each output place.

We can model services as Petri nets by assigning transitions to methods and places to states.[10] Each service has an associated Petri net that describes service behavior and has two ports: one input place and one output place. At any given time, a service can be in one of the following states: not instantiated, ready, running, suspended, or completed. After we define a net for each service, com-

position operators perform composition: sequence, alternative (choice), unordered sequence, iteration, parallel with communication, discriminator, selection, and refinement. These operators guarantee the closure property. Thus, by composing two or more Web services, we produce another service.

Let $\otimes$ be a sequence operator, for example, and $||_\alpha$ be a parallel operator with communication. We can then write our example as $A \otimes (B ||_\alpha C)$. We use a parallel operator with communication to compare and select between outputs of services *B* and *C*. Graphically, our service would look like Figure 1.

After specifying composition with a Petri net, we can use it to prove some algebraic properties, such as absence of deadlocks or livelocks (whether composition will terminate in a finite number of steps). Correct termination is very important for composed service; we verify this property by determining whether the Petri net is live and bounded.

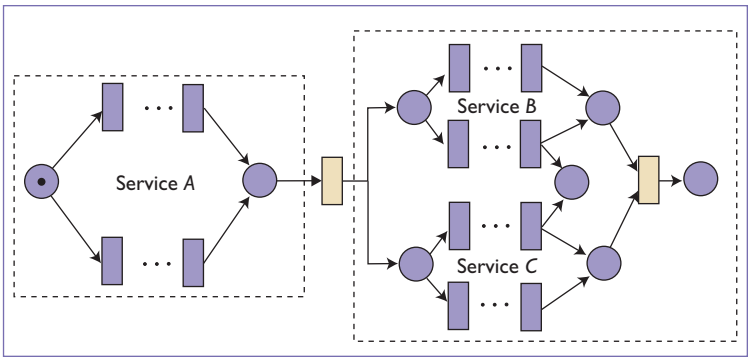

Figure 1. Petri net composition. To connect Petri nets representing services B *and* C*, we use a parallel operator with communication, and then connect this composition with a Petri net representing service* A *using a sequence operator.*

Model Checking and Finite-State Machines

Other approaches for Web service composition include model checking, modeling service composition as Mealy machines (described below), and automatic composition of finite-state machines (FSMs).

Model checking is used to formally verify finite-state concurrent systems. We describe system specification using temporal logic, then traverse and check the model to see whether the specification holds. We can apply model checking to Web service composition by verifying correctness inside a workflow specification. Among the properties we can check are data consistency, unsafe state avoidance (deadlock), and business-constraint satisfaction.[11]

Researchers have also proposed the *conversation specification* for Web service composition.[12] According to this approach, understanding constituent services' local behavior and the composed service's global behavior are important to verifying and guaranteeing correctness. The approach models services as Mealy machines, which are FSMs with input and output. Services communicate by sending asynchronous messages, and each service has a queue. A global "watcher" keeps track of all messages. The conversation is introduced as a sequence of messages. By studying and understanding conversation properties, the method provides new approaches for designing and analyzing "well-formed" service composition.

Automatic Web services composition is the ultimate goal of most composition efforts. Berardi and colleagues present a framework that describes a Web service's behavior as an execution tree and then translates it into an FSM.[13] They propose an algorithm that checks a composition's existence, and returns one if it exists. In the process, the composition is proved correct and the algorithm's computational complexity characterization is given, ensuring that the automatic composition will finish in the finite number of steps.

Method Comparison

We can now compare the various solutions with respect to our four service-composition requirements. We also discuss the possibility of automatic service composition. Table 1 summarizes our results.

Connectivity and Nonfunctional Properties

All approaches offer services connectivity. Although the services themselves are modeled in various ways, at the lowest level, the connection comes down to mapping and orchestrating input and output messages between the partner services' service ports. Most approaches neglect specification of nonfunctional QoS properties such as security, dependability, or performance. Only OWL-S lets users define some nonfunctional properties (namely, quality of service), but that capability has yet to be fully specified.

Composition Correctness

Verifying correctness depends on the service and composition specifications. BPEL and OWL-S provide no way to verify correctness. BPEL is a Turing-complete language dealing more with implementation than specification, and thus it's difficult to provide a formalism to verify the correctness of BPEL flows. All other approaches support verifica-

Table 1. Comparing service composition requirements.

	Service connectivity	Nonfunctional properties	Composition correctness	Automatic composition	Composition scalability
BPEL	√				Average
OWL-S	√	√			Average
Web components	√		√		Low
π-calculus	√		√		Good
Petri nets	√		√		Low
Model checking/FSM	√		√	√	N/A

tion in some way. Even OWL-S, when combined with Prolog or Petri nets, allows reasoning about correctness. However, the extent to which correctness is verified varies.

Web components offer a simple way to check for compatibility and conformance. π-calculus offers powerful algebraic verification for determining liveness, security, and quality of service. However, applying such verification depends on what is typed when you model services as processes. Petri nets use elaborate algebra for verification. We can check whether composition has deadlocks by determining whether the corresponding Petri net is live and bounded. Model checking's verification methods are comparable with π-calculus. Many methods are available for proving that a composed service's specification conforms to the model. The issue is deciding what needs to be specified for model checking to produce useful results. Another problem is computing resources (such as CPU time or storage space); given the vast state space you must examine, you can run out of resources and still not know whether the composition conforms to the model.

Automatic Composition

Many composition approaches aim to automate composition, which promises faster application development and safer reuse, and facilitates user interaction with complex service sets. With automated composition, the end user or application developer specifies a goal (a business goal expressed in a description language or mathematical notation) and an "intelligent" composition engine selects adequate services and offers the composition transparently to the user. The main problems are in how to identify candidate services, compose them, and verify how closely they match a request. So far, modeling services as FSMs is the most promising automatic composition approach.

Composition Scalability

All composition approaches support Web services connectivity through message passing via ports. Composing two services, however, is not the same as composing 10 or 100. In a real-world scenario, end users will typically want to interact with many services — consider the classic holiday booking scenario — while enterprise applications will invoke chains of possibly several hundred services. Therefore, one of the critical issues is how the proposed approaches scale with the number of services involved.

In BPEL, multiple service composition is somewhat tedious because XML files start to grow. Because BPEL composition is recursive, we can modularize composition. Unfortunately, BPEL has no standard graphical notation. Some orchestration servers offer graphical representation, and there are proposals to use UML-like notation for descriptions. Graphic notations are not formal, however, and they don't map one-to-one to BPEL's complex language constructs. OWL-S has similar issues.

The Web component approach achieves good scalability with class definitions, but requires additional time for mapping and synchronization between class definitions and XML. The π-calculus approach offers concise notation with powerful reduction mechanisms, which facilitate specification of complex services. The Petri net approach's scalability is reduced by complexity issues, since Petri nets are not a very scalable modeling technique. Finally, judging the scalability of model checkers and FSM models depends on the checker type and machine state operations. This discussion is outside our survey's scope, but with careful modeling, it's likely that a model checkers' scalability will be better than Petri nets and comparable to π-calculus.

Conclusion

Service composition approaches range from those aspiring to become industry standards (BPEL and OWL-S) to more abstract methods. An ideal

approach would cover all four key requirements that we identified. The main problem with "industrial" approaches is correctness verification. Service composition is sometimes called "programming in the big," yet it seems that industry is unaware that even "programming in the small" is plagued by numerous problems when formal specification and verification are lacking. We can't expect an open paradigm with such varying granularity as Web services to succeed based on implementation languages alone. On the other hand, formal approaches are often difficult to apply in real-world enterprise environments, and some face scalability problems. From the correctness viewpoint, it's beneficial to analyze Web service properties using elaborate mathematics; however, to realize these benefits, we must be able to translate from WSDL and SOAP to elegant mathematical solutions.

The Web service composability problem will likely be around for a while. Given this, our short-term goal should be to adopt an industry standard. A long-term goal must be to incorporate verification mechanisms that both scale well and let developers and users perform everyday chores using Web services — without having to worry about whether the process will deadlock, consume all the memory, disclose confidential corporate data, or send a credit-card number to an unknown recipient.

References

1. M.P. Papazoglou and D. Georgakopoulos, "Service Oriented Computing," *Comm. ACM*, vol. 46, no. 10, 2003, pp. 25–28.
2. 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.
3. F. Curbera et al., "The Next Step in Web Services," *Comm. ACM*, vol. 46, no. 10, 2003, pp. 29–34.
4. A. Ankolekar et al., "DAML-S: Web Service Description for the Semantic Web," *Proc. Int'l Semantic Web Conf.* (ISWC), LNCS 2342, Springer-Verlag, 2002, pp. 348–363.
5. S. McIlraith and T.C. Son, "Adapting Golog for Composition of Semantic Web Services," *Proc. Int'l Conf. Principles of Knowledge Representation and Reasoning* (KRR 02), 2002, pp. 482–493.
6. S. Narayanan and S. McIlraith, "Simulation, Verification and Automated Composition of Web Services," *Proc. Int'l World Wide Web Conf.* (WWW2002), 2002, pp. 77–88.
7. J. Yang and M.P. Papazoglou, "Web Component: A Substrate for Web Service Reuse and Composition," *Proc. 14th Conf. Advanced Information Systems Eng.* (CAiSE 02), LNCS 2348, Springer-Verlag, 2002, pp. 21–36.
8. R. Milner, "The Polyadic π-Calculus: A Tutorial," *Logic and Algebra of Specification*, F.L. Bauer, W. Brauer, and H. Schwichtenberg, eds., Springer-Verlag, 1993, pp. 203–246.
9. L.G. Meredith and S. Bjorg, "Contracts and Types," *Comm. ACM*, vol. 46, no. 10, 2003, pp 41–47.
10. R. Hamadi and B. Benatallah, "A Petri-Net-Based Model for Web Service Composition," *Proc. 14th Australasian Database Conf. Database Technologies*, ACM Press, 2003, pp. 191–200.
11. X. Fu, T. Bultan, and J. Su, "Formal Verification of E-Services and Workflows," *Proc. Workshop on Web Services, E-Business, and the Semantic Web* (WES), LNCS 2512, Springer-Verlag, 2002, pp. 188–202.
12. T. Bultan et al., "Conversation Specification: A New Approach to Design and Analysis of E-Service Composition," *Proc. Int'l World Wide Web Conf.* (WWW 2003), ACM Press, 2003, pp. 403–410.
13. D. Berardi et al., "Automatic Composition of E-Services that Export Their Behavior," *Proc. 1st Int'l Conf. Service-Oriented Computing* (ICSOC 03), LNCS 2910, Springer-Verlag, 2003, pp. 43–58.

Nikola Milanovic is a research fellow and PhD candidate at the Institute for Informatics, Humboldt University, Berlin. His research interests include component- and service-based environments, service composition, ubiquitous computing, ad hoc networking, and wireless communication. Milanovic received a Dipl. Ing. in electrical engineering from the University of Belgrade. Contact him at milanovi@informatik.hu-berlin.de.

Miroslaw Malek is a professor and chair of computer architecture and communication at Humboldt University, Berlin. His research focuses on high-performance responsive computing, including parallel architectures, real-time systems, networks, and fault tolerance. Malek received a PhD in computer science from the Technical University of Wroclaw, Poland. He is a member of the ACM. Contact him at malek@informatik.hu-berlin.de.