WeSCoS: A First Step Towards Web Programming

Laurence Melloul
Computer Science Department
Stanford University
Stanford, CA 94305
melloul@cs.stanford.edu

Trevor Pering
Intel Microprocessor Research Labs
5350 N.E. Elam Young Parkway
Hillsboro, OR 97214-6497
trevor.pering@intel.com

Armando Fox
Computer Science Department
Stanford University
Stanford, CA 94305
fox@cs.stanford.edu

Abstract

The characteristics of the Web and its services, such as typeless data, service accessibility, and service autonomy, provide an appealing environment for composing services, although these same properties raise new challenges. We have prototyped WeSCoS, a Web Service Composition System, that is itself accessed as a Web service and provides a framework for composition. It is an early testbed to experiment with Web composition and identify distributed system issues that will need to be addressed in order to enable a style of large-scale Web programming, in which developers will use Web services as simple components to be loosely integrated into robust, large-scale applications. The main design component of WeSCoS is a per-service specification document, which encapsulates all necessary information for the WeSCoS system to automatically execute service requests, both atomic and composite.

1 Introduction

WeSCoS (Web Service Composition System) strives to leverage the vast array of existing Web services by enabling service reuse in a manner similar to that of component-based software development. WeSCoS encourages software developers to treat Web services as simple software components used to build larger, distributed Web applications, rather than as monolithic entities only. In WeSCos, composable entities, consisting of both atomic services and compositions, are described in a self-contained service specification. When executed, composition specifications ultimately trigger basic Web service interactions, which are then combined to produce the aggregate result.

A simple example of a composed service supported by the current system is “find restaurants near theaters in a given zip code,” which involves 1) finding theaters in a given zip code, 2) converting each theater’s address to latitude and longitude, 3) locating restaurants near that location, and 4) combining all the results. Other example compositions include sorting hotels by the number of nearby restaurants, obtaining restaurant reviews from multiple sources, ordering groceries based on a list of recipes, and the canonical comparison shopping. Although all of these examples can be implemented without WeSCoS using ad-hoc scripting techniques, a supporting framework will facilitate their development, enabling the creation of more complex services with lower overhead.

The next section presents the motivation behind using Web services as distributed building blocks, discussing their advantages and disadvantages in detail. Section 3 covers the implementation of the WeSCoS service specification and execution environment. Related work, covered in Section 4, focuses mainly on service composition in traditional object-oriented environments and also covers related technologies in the Web domain. Section 5 discusses future work: how the composition framework can be extended to aid the construction of large-scale distributed systems by automatically supporting basic distributed system mechanisms.

2 Motivation

Just as reusable software allows new systems to be rapidly created by composing independent modules, so the large selection of services on the Web makes it appealing to use them as building blocks in the creation of larger and more sophisticated services. Composition enables the creation of large systems because it supports reuse: previous work, performed by independent, specialized organizations, can be incorporated into new systems, decreasing the implementation cost. Additionally, Web services represent vast stores of data that cannot be otherwise centralized, because the data is proprietary, too large in volume, or too frequently changing to make warehousing practical. Beyond code reuse, Web service composition allows us to leverage these databases.

A Web composition *framework* facilitates the composition of Web services by streamlining the incorporation of existing services and enabling inter-service composition. Web developers inject their services into the system, immediately making them available for composition. In case they specify a composite service, the system creates a new Web service, in turn available for composition, and becomes responsible for automatically executing every step, from invoking the composed services to appropriately assembling the results, upon user requests.

We describe the benefits and challenges of Web service composition as compared with traditional object-oriented software module composition, outline an architecture that exploits the strengths and addresses some of the challenges of Web service composition, and discuss experience with our early prototype implementation. Although this is early-stage work, our larger goal is a research agenda leading to an Internet in which software is routinely built by on-demand composition of autonomous services. Hence the primary goal of our prototype is to illuminate the challenges for ongoing work and provide a platform on which we can conduct future experimentation for large-scale Web programming.

2.1 Web Services Make Good Building Blocks

Web services are good building blocks for large-scale applications because they are publicly accessible, homogeneous, use fluid calling conventions, and are autonomous:

1. **Accessibility,** afforded by the public knowledge of the service interface increases the number of basic services available for composition. This information, such as the service script's URL or HTTP method, is currently encapsulated in the HTML pages associated with the service. In contrast, in traditional object-oriented composition, composers and service providers must agree on a location where to make service client stubs available, hence making the injection of new services in the system dependent on service provider's will and reactivity.

2. **Homogeneity**, meaning all **-** or most of – Web services are accessed through HTTP, makes it easy to interact with all services uniformly; otherwise, the implementation overhead caused by composing autonomous services deployed over different, but equally dominant, protocols such as Java RMI, DCOM and CORBA ORBs (Object Request Brokers), becomes prohibitive.

3. **Calling conventions:** Web services manipulate typeless information. All data is passed as strings, with descriptive keyword names (rather than datatype names) used to identify arguments in invocations (form submissions). This has two important side effects: first, information can be uniformly transferred between services, without requiring casting or the use of conversion functions, as would be necessary if using strongly-typed interfaces in object-oriented software composition [1]; second, because HTML forms use keyword-based argument passing (also found in scripting languages such as Tcl [2] and Perl [3]), invocations are generally resilient to the number and order of arguments, avoiding some brittleness when the service evolves. (RPC marshalling, while more efficient, requires observance of the number of parameters and their ordering.)

4. **True autonomy** keeps Web services clearly decoupled from one another: initially developed and maintained independently, services are not altered by type assumptions about their counterparts or interoperability code changes. In addition to providing excellent fault isolation, true autonomy makes no single entity responsible for the *entire* system: each individual service undertakes the responsibility for development, availability, and robustness of its own functionality. This is an extremely important consideration to create a new large system because otherwise the coordination and robustness of a large monolithic project becomes impractical.

2.2 Web Service Composition Presents New Challenges

Although autonomous services provide many desirable characteristics, they also present many challenges.

1. **Lack of apparent structure.** Web services today return results that are not easily machine-readable: their structure is not apparent because the results were generated for human interaction. Without a mechanism for incorporating new services into the system, a custom wrapper for each individual service is required, making the cost of incorporating such services prohibitive. In the future, we expect the move to XML-based Web service representation to alleviate this problem.
2. **Lack of guarantee.** Because autonomous services are independently developed and maintained, a service consumer (or composition engine) has very little control over the quality and availability of the foreign service. For example, if a bug is found in a remote service, a consumer has no way to encourage speedy repair. In practice, the market economy of Web services, revolving around the principle of competition, necessitates these services to do the best they can at meeting their customers needs, so many of the issues are likely to be solved "by popular demand."
3. **Lack of support for automatic composition validation.** As we describe later, the lack of typing of Web data removes the benefit of compile-time automatic error detection (although it has the advantage of not discarding logically valid compositions) and places the burden of validation on the developer creating the composition. Manual verification may be acceptable for small-scale projects, but it is a liability for larger systems.
4. **Wide-area distribution.** Autonomous Web services are remote, which means that high-bandwidth or low-latency transactions are not always possible, although current ASP[1]-style collocation and peering technology should alleviate this.
5. **Independent evolution.** The last and most difficult problem with autonomous services is that a user or composition engine has no control over the evolution of the service interface: a service can change its interface's key elements without notice, perhaps causing composition systems to break.

In summary, although new challenges arise that are unique to (and result from) the Web's HTTP protocol and loosely-coupled, widely-distributed nature, a Web service composition framework, for that same reasons, will help solve many of the traditional problems of object-oriented composition systems. We believe Web services' true autonomy makes the Web a promising environment for reuse and large-scale composition.

3 The WeSCoS Architecture and Execution System

The WeSCoS system allows developers to add services into the system and specify composition requests. It is itself a Web service. It is centered on the concept of a *service specification*, a single document, written in XML (eXtensible Markup Language) [4], that characterizes a remote autonomous Web service and encapsulates all the information required to invoke against it and collect

[1] Application Service Provider

and parse the results. The same specification interface is used to specify an atomic service or a composite service, making these appear identical to the composition engine and its users. Once a service specification is loaded into WeSCoS, the service is made available for composition on the Web.

Compositions in the service specification are specified using XPL (XML Programming Language), a scripting language we have defined to facilitate composition definition and processing.

The WeSCoS execution system, shown in Figure 1, implements the processing of WeSCoS service specifications to execute user requests. The architecture consists of the Service Processing Core, a Database to store specifications, and a Web Front-End to enable users to invoke services through a Web browser. When a request is made, the system fetches the XML specification from the Database and forwards it through the appropriate stages, annotating it with additional information as it passes along. The final document is a complete characterization of the request, including the initial specification, input fields, results, and logging information.

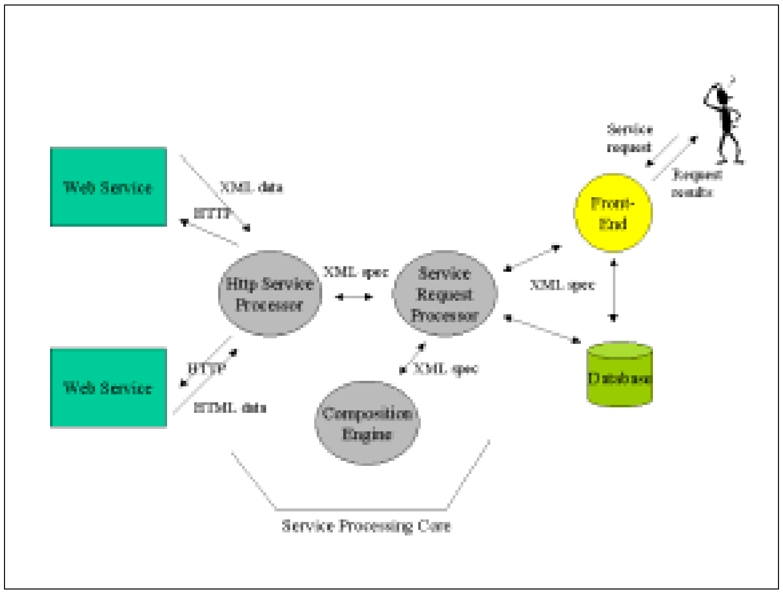

Fig. 1. The WeSCoS System architecture consists of three main logical components: the Web front-end, the service processor core, and the database.

The front-end is responsible for two major interactions: submitting a new request, and displaying the results of a completed request. First, the user selects a request from a simple list of available services; the front-end then retrieves the specification from the Database and automatically generates a HTML form based on that specification to get user input. When the user submits a completed form, the front-end inserts the input values into the XML specification and passes it to the Service Request Processor. After the system completes the composition, it annotates the document with the service invocation results, from which the front-end generates HTML output.

The Service Request Processor routes a service request to the appropriate destination based on the nature of the service specification: to either the HTTP processor or composition engine. The HTTP processor retrieves the appropriate contact information from the specification, interacts with the external Web-server, and places the parsed result in the document. The composition engine executes the composition steps, that is to say the XPL statements specified in the specification, creating output as it progresses. When it encounters an invocation statement, it forwards the request to the service request processor which in turn handles the new request adequately.

3.1 Design of the WeSCoS Service Specification

The WeSCoS service specification is critical for the WeSCoS system as it includes all necessary information for the system to automatically process user requests. It permits users to both describe an HTTP service or specify a composition request. Several key design points are fundamental to the specification design and the WeSCoS system; they basically made it possible to design a flexible execution composition framework.

1. A separate specification document

For WeSCoS to be a composition framework, all variant information, which are specific to the services (service url, input/output parameters, etc.), are left outside the execution system, available and accessible when services need to be invoked or compositions to be processed to fulfill a user request. For that reason, the service representation is a separate specification document, which is loaded for use at run-time only. The service invocation and composition mechanics are in the system, but the details specific to the service are in the specification. The specification is the only source of information needed to invoke against a service, and the only place in which changes due to service evolution need to be reflected.

2. An XML specification

We identify the three following requirements for the service representation:

- *Typeless, keyword-identified data*

  We already identified the Web service features such as typeless data (mainly strings) and keyword-based argument passing that make Web composition appealing. We would like our service specification to preserve these properties: it must allow the encoding of Web data as sets of self-describing, typeless, keyword-identified parameters.

- *"Typeless" data structures*

  For easier information retrieval and processing, we would like Web service interface parameters to be described using *some* structure, without however imposing rigid types (really data structure names). Dissociating the type name from the structure of the element permits compositions of truly autonomous services, that might create different types for the same data structure, or use a same type name for different data structures.

- *Attach information to leaf elements of data structures*

  A third requirement of the service specification language is the ability to attach identifying keywords not only to data structures, but also to individual elements within those data structures. The motivation is that frequently a Web service will return several result fields from an invocation (e.g. an address-lookup service might return name, address, state, and zip), and it may be desirable to use only *some* of those fields for composition with a second service (e.g. a zip code to ZIP+4 mapper). Such compositions are possible only if we expose the leaves of the data structures.

Given these requirements, we selected XML as the medium for service specification. As the emerging representation of choice for semi-structured data, XML allows the definition of the input/output data structures of the service without imposing rigid types, still providing enough structure for automatic data processing. Furthermore, because XML expressions are nodes in a tree, it is easy to access sub elements of the input/output parameter's data structures within the service specification through tree path expressions. (This is the approach adopted by databases for semi-structured data, such as Lore [5].) Unlike class definitions in object-oriented languages, no access methods are needed and there is no issue with private or protected access to data members: in the XML service specification, all elements that are exposed are essentially public.

3. A new scripting language to specify compositions

XPL is the XML Programming Language we defined for specifying composition instructions in the service specification. XPL, which we describe in section 3.4, is an interpreted language with very limited capabilities such as iteration and set selection; it is adequate for simple compositions. Since XPL is implemented using the same general syntax, XML, as the overall service specification, and is simple to use, it does not necessitate learning another syntax or language, perhaps more complex. In addition, not embedding another scripting language such as Tcl restricts the type of instructions that are executed by the system, and limits its set of unpredictable behaviors such as infinite loops. Finally, compositions expressed in XPL are simple enough that the speed of execution and restricted nature of the language do not hinder the system. (We recognize however that it is hard to impose a new language, how easy it is to learn; XPL was mainly designed for a research purpose, and nothing prevents the extension of the specification to incorporate other languages, as long as the execution system understands how to process them.)

4. A uniform interface for "atomic" and composite services

We would like useful compositions to become new building blocks themselves. To this end, execution details are isolated in one section of the representation, held separate from the service identification and interface description information. This allows uniform interaction with all specifications, regardless of the type of the service (atomic or composite), and makes it easier to understand the service by clearly separating the service interface from implementation details. It also limits the impact of a change of service definition: changes in the execution information (such as service URL) only impact one specification; changes in the description information (such as new input parameter) might also impact specifications that compose this service.

3.2 A Simple Service Description Example

Figure 2 gives an example of specification for the atomic, "CD search by author" service using Amazon.com. There are four major components of the document: an identifier, input schema, output schema, and execution information.

```
<service version = "1.0">
    <identifier>
       <name>Get CD list by author</name>
       <provider>Amazon.com</provider>
       <version>1.0</version>
       <description>Return the list of CDs, with title and price,
                    for given author</description>
    </identifier>
    <input-schema>
       <element name="field-keywords" desc="authorName"
                type="String"/>
    </input-schema>
    <output-schema>
       <element name="result" minOccurs="0" maxOccurs="unbounded">
       <element name="title" desc="CDTitle" type="String"/>
       <element name="price" desc="CDPrice" type="float"
                minOccurs="0"/>
       </element>
    </output-schema>
    <execute method="http" version="1.0">
    …
    </execute>
</service>
```

Fig. 2: The Service Specification, version 1.0. The "CD search by author" example using Amazon.com. There are four major components: the identifier, input-schema, output-schema, and execute nodes.

The <identifier> node contains all the information necessary to describe the function, information source, and version of the service. Input and output schemas describe the interactions the specification is able to support; they are specified in a loose subset of XML-Schema [6]. Each <element> tag represents a field in the service's input/output with the name specified (in the example, "field-keywords" is the name of the parameter expected by the HTTP method to call the service). Currently, elements can either be simple, like "field-keywords," and contain data, or complex, like "result," and contain other elements. Additionally, nodes can be tagged as repeating, with "maxOccurs," or optional, with "minOccurs." Schemas are used to describe the expected input and output parameters, both for the composing system, and the composer through the descriptive "desc" field. They also let specify some type information, for possible use by strongly-typed service backend infrastructures.

The bulk of the differentiating information for a service is contained in the <execute> node. This node has a method indication and a version number. The methods currently supported are "http" for the invocation of an atomic, HTTP service, and "comp" for the execution of a composition request. (At the expense of more complexity in the composition process, the execute node could be extended with other methods, such as <execute method="tcl"> or <execute method="java">.)

The <execute> node version number indicates the version of the method (not to be confused with the HTTP version itself in the case of the "http" execute node), necessary to the processing system, in case of further evolution of the method.

3.3 An Execution Example: The "http" execute node

The "http" execute node, shown in Figure 3, contains information to invoke an HTTP service and parse the result document. There is basic contact information, primarily the service URL, and parsing patterns, which can either be complex in the case of HTML-returned documents, or trivial if the result document is returned in machine-readable XML (in that case, the output-schema node provides sufficient information for result extraction).

```
<execute method="http" version="1.0">

    <url>http://www.amazon.com/exec/obidos
    /search-handle-form/102-0977834-
    7085735</url>

    <method>post</method>

    <version>1.0</version>

    <parse method="html">

       <result>

        <![CDATA[<tr…<td…<a…>title</a>…~au
thor<…Our Price:…$price<…</tr>]]>

      </result>

    </parse>

</execute>
```

```
<execute method="http" version="1.0">

    <url>http://www.amazon.com/exec/obid
    os/search-handle-form/102-0977834-
    7085735</url>

    <method>post</method>

    <version>1.0</version>

<parse method="xml"/>

</execute>
```

Fig. 3: The "http" execute node, for services returning respectively HTML and XML results. It contains all necessary information for the HTTP method call and result parsing. Parsing can either be accomplished using HTML, which requires pattern-matching information, or XML.

3.4 The "comp" execute node and the XPL language

The "comp" execute node (Figure 4) specifies what services to invoke and how to merge results, using XPL (XML Programming Language). The primary use of XPL is to invoke services (<invoke> instruction), iterate over results (<foreach>), and create output information in a document (<add>). XPL is built on top of XPath [7], which selects components of an XML document using tree expressions. The <invoke> and <foreach> operations bind variables which can be used as the root of an XPath expression. For example, <foreach bind="res" data="A/result"> iterates over all the *result* elements of *A*, binding each one to the variable *res*; "res/title" then access the appropriate title element of *A*.

The <invoke> operation is the most important in WeSCoS because it provides the basic mechanism for composition: it is responsible for invoking another service representation, selected by the encased <identification> node, and for mapping parameters as appropriate. When nested inside of a loop, <invoke> can be used to iterate calls to other services with variations in the input parameters. Additionally, <invoke> can be used to transparently call other composite services.

<add> creates a hierarchically structured output element in the service document, eventually displayed in HTML by the Front-End or used by another composition request.

```
<execute method="comp" version="1.0">
      <invoke bind="A">
          <identification>
              <name="Get CD list by author"/>
              <provider="Amazon.com"/>
              <version="1.0"/>
          </identification>
          <param name="field-keywords" source="input/author"/>
      </invoke>
      <foreach bind="res" data="A/result">
          <invoke bind="B">
                <identification>
                   <name="Get CD info by title"/>
                   <provider="Buy.com"/>
                   <version="1.0"/>
                </identification>
                <param name="qu" separator=" ">
                                <value source="res/title"/>
                                <value source="res/author"/>
                </param>
          </invoke>
          <add element="result">
                <add element="author" source="res/author"/>
                <add element="title" source="res/title"/>
                <add element="price1" source="res/price"/>
                <add element="price2" source="B/result/price"/>
          </add>
     </foreach>
</execute>
```

Fig. 4: The "comp" execute node provides instructions on how to actually compose services. This example contacts Amazon.com and in turn invokes Buy.com on each result received. Results from Buy.com trigger the insertion of a result node that combines data from both sources.

As an example of nested compositions (not shown in this paper for space limitations), we specified a service that, given a zip code, gets the list of theaters with playing movies and show times, along with the list of nearby restaurants. For that purpose, we first created a service that gets the list of restaurants near an address, which composes the geocode.com service that provides latitude and longitude information of a given address, and the mapquest service that provides restaurant information around specific latitude/longitude coordinates. Next we composed the moviefone.com service, which provides the list of theaters and their addresses, with that newly created service, calling it on each theater address. For one user invocation, this service automatically executes more than 10 invocations, for an average of five theaters found.

3.5 Initial Evaluation

WeSCoS is an initial step in identifying a fruitful research agenda in service composition. As an initial prototype, its implementation is sparse and necessarily limited. However, the prototype has yielded some immediate insights into the construction of a composition framework. In particular, our initial experience confirms some of our hypotheses about the desirability of using the Web as a composition substrate, and suggests some alterations that should be made to prevent unnecessary difficulty later on.

- **Ease of creation:** It is relatively easy to add a new service composition into the system, which was part of the initial design goal. This is because the entire specification is contained in one file, and XPL is simple enough to use to specify compositions.

- **Input/output validation:** One difficult step for a user to create a new composition specification is matching input or output data of the services referenced in the <execute> node with its associated schema in the composed service specifications. Useful tools to help with this would be a validating XML parser to check the results or input, or a validating GUI that would also guide users selecting services.

- **Loose type mismatches:** Even though two interfaces may specify services between which passing a particular parameter is *semantically meaningful*, this does not necessarily mean that the parameter's format will be *syntactically compatible*. For example, some services specify results in all capitals or do not include punctuation, which makes strict comparisons with other results problematic. It is cumbersome to perform the necessary translations using a procedural language such as XPL or within the execution system; the use of regular expressions would make this easier.

- **Parsing HTML:** By far the most difficult aspect of creating a new basic HTTP service is parsing the return HTML document, for three reasons: first, production HTML is complex and varies in its compliance with the HTML specification, making it difficult to create an accurate and robust parsing expression; second, service providers frequently change the output format of the "decoration" parts of their pages; third, client side scripts such as JavaScripts or ActiveX scatter non-exploitable information all over the document. Parsing HTML is not a core part of the WeSCoS service: it is merely a temporary means to provide services for composition, for which we can leverage useful tools such as WebL [8] and Perl. Although we will likely investigate how to detect relevant changes within HTML documents, we anticipate that increased adoption of XML by Web service providers will make this problem less relevant over time.

- **UI Generation:** The current automatic user-interface generation mechanism is adequate for interacting with simple services, but the display becomes quite complex for results with multi-level detail. This behavior is sufficient for a research platform, but suggests research into developing better user-interface generation for ad-hoc services.

- **Performance:** The prototype is quite slow (minutes) in performing compositions, for two reasons: first, each HTTP transaction requires a round-trip to a remote server, and our prototype operated behind a firewall with limited capacity; second, the prototype code is naïve—it does not exploit parallelism inherent in a composition, and the HTML parsing implementation is quite slow. In a modern peering infrastructure, we expect composition time to be dominated by server latency at each destination Web server.

4 Related Work

There are several techniques possible for connecting two autonomous services, most notably using a global type system similar to strong types found in many programming languages, or using flexible types, which set no *a priori* restrictions on how the data should be used [1], and put the onus for validation on the developer. A global type system aids composition by requiring interfaces to type-match, preventing incorrect compositions. However, such a type system requires a standardization process by which all composable entities agree on the type hierarchy, which is acceptable for well-behaved or centralized domains but can be problematic in distributed domains when types are added or changed. The problem is clearly exacerbated on the Web because of the independent evolution and development of Web services. Ousterhout [1] argues for the use of a loose type system, late type checking, and keyword-based passing of string arguments (as opposed to marshaling and unmarshaling with a strong type system) as techniques for *integration-based programming*. Roughly, integration-based programming consists of using a scripting (integration) language to glue together autonomous components into useful larger systems; the lack of strong types in integration-based languages means that the type system places no *a priori* restrictions on which inter-entity calls are allowed, making the system more robust in the face of type evolution. Web composition clearly has much in common with this vision, and we adopt Ousterhout's recommendation in our service specification. Recent work on flexible type systems [9] proposes specific techniques by which the type system can be used to validate certain compositions while still allowing composed services to evolve independently; we are investigating the extent to which those ideas can be integrated into WeSCoS. It is likely, therefore, that a hybrid approach, which incorporates the right aspects of both systems, is necessary to handle the composition of Web services. The WeSCoS service specification, by providing for type specification, should offer enough flexibility for such a hybrid system.

Composition systems such as CHAIMS [10] provide a framework to compose autonomous services, deployed mainly in object-oriented infrastructures but extensible to the Web. CHAIMS services must provide wrappers that comply to CPAM (CHAIMS Protocol for Autonomous Megamodules), a particular invocation protocol whose primitives include service cost estimation and service progress checking. Imposing this requirement results in a compilation step for CHAIMS systems, but these functions are compelling and useful when composing autonomous services. We believe that eventually, a composition framework should benefit from these. HADAS [11] and Ninja [12] include a "conversion" wrapper, collocated at the composition engine (client), to make all services accessible over Java RMI. In these systems, a change in the definition of a service is reflected in code rather than in a declarative specification, and strong typing makes compositions brittle because any change in the type system causes potentially valid compositions to be rejected.

On the Web, service aggregators such as CNET.com or Yodlee.com compose services, but they do not expose the composition framework to end users, and it is unclear whether their compositions are ad-hoc or based on a systematic framework. They do not appear to have the goal, as WeSCoS does, of allowing large-scale systems programming on the Web.

Various XML tools or specifications, a couple of which described below, help for Web service invocation. They also intend to facilitate communications between services, by providing naming standards (Rosettanet.org) for easier transfer of information between services. SOAP [13] (Simple Object Access Protocol) and SCL (Service Contract Language) are two emerging XML standards that provide access to Web-based services. SOAP is a mechanism to invoke a remote method through XML, making it possible to define the scope of a method within a larger functional module. In the WeSCoS system, a service specification can describe a SOAP method, and it would be relatively easy to add a <execute method="SOAP"> node in the XML specification to have the system invoke SOAP services when they become available. SCL is potentially very similar in function to the WeSCoS service specification: it is an XML interface for SOAP services. At the time of this writing the SCL draft is incomplete, so it is too soon to tell how much overlap the two systems will have; however, the current SCL draft does not suggest it is targeting self-contained composition of services.

5 Future work

WeSCoS currently supports simple services only. We plan to extend the set of services to more complex ones, such as services that require multiple sessions (e.g., through multiple forms to fill in), services that have side-effects (e.g., through executed payments), or services that present security enhancements (such as login and password). Multi-session services can be treated through simple composition specifications; side-effect services are more delicate to handle within composition, as they may require some degree of transactional semantics. Dealing with error recovery and rollback in these cases will obviously present a challenge.

We will complement the WeSCoS front-end to allow Web users to add new service specifications. Besides being able to validate specifications, we will have to address the issues of service classification and selection to make the service repository scalable. The service repository could itself be a separate Web service, possibly replicated. A service specification discovery mechanism would be necessary. We are still investigating the best interface to inject and maintain services in the system. We believe a community approach will benefit both users and the WeSCoS system, by distributing the responsibility of defining and maintaining specifications.

Because composition reduces the number of user actions to invoke a complex task, it will be particularly useful for small-glass device interaction. Past work on small-device Internet access [14] suggests that it should be straightforward to develop XML-based interface converters to provide access to the composition system through small-glass devices.

It will also be nice to offer a GUI and abstract the necessary steps to define a service or composition request; that way, non-technical Web users could also directly benefit from Web composition.

When composing autonomous services, issues such as error recovery, invocation synchronization, and security will need to be tackled in newly interesting ways:

- **Error model and error recovery:** How do we identify an error whose source is in a process we do not control, and how do we recover from it?

- **Invocation synchronization:** How do we handle unpredictable timeouts due to nested composition? This is very likely to happen if multiple WeSCoS system instances simultaneously serve different sets of composition requests, which automatically become available for composition by the other WeSCoS instances, and so on. When a service is invoked, it might not be known from the WeSCoS caller system at what depth in the composition the error originally occurred, and how the system should react to invocation latencies. (This problem is related to the problem of graceful exception handling in component-based OOP.) Also, how can we detect and recover from cycles in the composition graph or mutual recursion?

- **Security:** Because of the lack of control over the services, it is critical that the composed services must be trusted, or that the composition system has a way to verify the adequacy of the results returned by one service, before transferring them to the next service in the composition.

The above observations form at least some elements of a rich research agenda for Web composition, and offering users with opportunities to build large-scale applications through integration-based Web programming.

6 Conclusion

The WeSCoS framework enables web programming by providing a mechanism for composing autonomous web services. The salient properties of web services make them an ideal candidate for composition, presenting a wealth of distributed resources to encourage service development without prohibitive startup costs. An XML based service specification encapsulates all the information necessary to reuse a web service, supported by a run-time environment which handles front-end user-interface management and processing of the specification document. Initial experience with WeSCoS on small compositions indicate that it is an environment conducive to creating composite services, proving to be a viable platform for investigating complex properties of large-scale distributed web applications.

Although there is obviously much to be done, and the issues we have uncovered with our early prototype are barely sufficient to allow us to formulate a research agenda, we believe the potential impact of this approach is sufficiently wide that the challenges are worth addressing seriously. The decentralized nature of the Web, the simplicity and lack of structure in its protocols and formats, and the autonomy of evolution of Web servers and clients are simultaneously the characteristics that make Web-based composition both frustrating and tremendously promising compared to traditional object-oriented programming. We hope to enable rapid progress toward a Web in which most services are composed by default and in which large-scale sophisticated applications created by composition become as commonplace as HTML form-filling is today.

7 Acknowledgments

The research described in this paper has benefited from the support from Intel Corporation, and comments from several people, whom we would like to thank: Eric Anderson, Steve Bennett, George Candea, Mike Chen, Steve Gribble, Paul Pierce, Scott Robinson, and Nick Wade.

8 References

[1] J. Ousterhout. *Scripting: Higher Level Programming for the 21st Century*. IEEE Computer Magazine, March 1998.

[2] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, ISBN 0-201-63337-X, 1994.

[3] L. Wall, T. Christiansen, and R. Schwartz. *Programming Perl*. Second Edition, O'Reilley and Associates, ISBN 1-56592-149-6, 1996.

[4] http://www.w3.org

[5] D. Quass, J. Widom. R. Goldman, K. Haas, Q. Luo, J. McHugh, S. Nestorov, A. Rajaraman, H. Rivero, S. Abiteboul, J.D. Ullman, and J.L. Wiener. *LORE: A Lightweight Object REpository for Semistructured Data*. Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, Canada, June 1996. Demonstration description.

[6] http://www.w3.org/XML/Schema

[7] http://www.w3.org/TR/xpath

[8] http://www.research.digital.com/SRC/WebL/library.html

[9] A. Begel and M. Spreitzer. *Toward more flexible types*. In Proc. WET-ICE 99.

[10] L. Melloul, D. Beringer, N. Sample, G. Wiederhold. *CPAM, a Protocol for Software Composition*. CAiSE'99, Heidelberg, Germany, June 1999; Springer LNCS volume 1626.

[11] I. Ben-Shaul, A. Cohen, O. Holder and B. Lavva, HADAS: A Network-Centric Framework for Interoperability Programming, International Journal on Cooperative Information Systems, World Scientific Publishing Company, Vol. 6, Nos.3&4, 1997, pp. 293-314.

[12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer. *Adapting to network and client variation using active proxies: Lessons and experience.* IEEE Personal Communications (invited submission), August 1998.

[13] http://ninja.cs.berkeley.edu

[14] http://www.w3.org/TR/SOAP/