Enabling "Smart Spaces:" Entity Description and User Interface Generation for a Heterogeneous Component-based Distributed System

Todd Hodes, Randy Katz
Computer Science Division
University of California, Berkeley
Berkeley, CA 94720-1776
{hodes,randy}@cs.berkeley.edu

July 17, 1998

Abstract

This paper motivates and describes a document-centric framework for component-based distributed systems. In the framework, XML documents are associated with programs that provide either static, immutable interface descriptions as advertisements of functionality at the server-side, or specification of manipulations of these server descriptions to express their usage at clients. We illustrate how the framework allows for 1) remapping of a portion of an existing user interface to a new room control (for example, due to movement of the terminal) 2) viewing of arbitrary subsets and combinations of the functionality available, and 3) mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-centric framework in addition to a conventional object-oriented programming language provides a number of key features. One of the most useful is that it exposes program/UI to referent objects mappings, thereby providing a standard location for manipulation of this indirection.

1 Introduction

Work in "smart spaces" overlaps a number of related research areas — active networking [1], networking middleware [2, 3], advanced user interfaces [4], mobile computing [5], and "ubiquitous computing" [6], to name a few — while at the same time focusing on a novel new domain for integration and extension of such work. One of the most challenging new issues is coping with the inherent *application heterogeneity* caused by the need for a location to adapt to individual users rather than vice-versa. Specifically, the challenge is allowing dynamic adaptation and reconfiguration of end-user applications in response to changes in available functionality and modes of user interaction. If users can only interact with "conventional" software, they will be constrained in their ability to customize the space to the extent in which the designers of the applications in it predicted their needs and/or provided a programmatic interface. Most often, the degree of freedom is limited by the user interface: application state and application preferences can only be manipulated through it, and the user interface itself is not highly configurable. Such is the case with monolithic programs, which are notoriously deficient in providing hooks into application state or in making it accessible via a well-defined external protocol. (On the other hand, monolithic user interfaces provide ease-of-use for well-understood functionality.) Client/server programs fare better, in that they provide (by definition) a exposed protocol, but suffer two problems: the specification of the interface is often ad hoc, and only a programmer can make changes to their view of the functionality (by modifying the client code).

One approach to addressing these problems is to allow application programs to be downloaded on-the-fly to handheld devices and uploaded to local computers [7]; for example, as Java applets. The difficulty of this approach, though, is that it does not allow the end-user to customize applications for interaction with a heterogeneous set of services as related entities. In other words, it cannot overcome minor differences in protocol — even for functionally identical services — because the applications are opaque. For example, imagine needing to download a different application upon encountering every new light switch in the world in order to use it. Though the functionality is exposed, it is not in a form amenable to manipulation.

Another approach is to standardize functional interfaces and require that applications (and spaces) support these standards, thereby avoiding the above bullet. The difficulty here, though, is that enforcing a plethora of application-specific standards is impractical.

Clearly, a different model than either of the above is preferable, one that balances the need to expose interfaces with the need to agree on protocol standards. Herein, we propose such a new approach, leveraging a component-based application framework [8] and pairing it with an architecturally-independent document model for component descriptions. It hybridizes features of the two basic approaches discussed above, allowing downloading/uploading of code fragments (as specified by the documents) and imposing a standard for interface description and manipulation that is not application-specific.

This paper describes our initial thoughts and work on this infrastructure, a novel prototype of a document-centric framework for component-based distributed systems. In the framework, application programs and user interface programs are associated with documents that provide one of either

- static, immutable interface descriptions as advertisements of functionality, or
- specification of manipulations of these server descriptions to express their usage.

We describe the framework and illustrate how it allows for

- remapping of a portion of an existing user interface to a new room control (for example, due to movement of the terminal)
- viewing of arbitrary subsets and combinations of the functionality available, and
- mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-centric framework in addition to a conventional object-oriented programming language provides a number of key features, the most critical is that it exposes program/UI to referent objects mappings, thereby providing a standard location for users (and their programs) to manipulate these mappings.

The rest of this paper is structured as follows. Section 2 introduces our document-centric component framework design. Section 3 describes the investigation approach. Section 4 introduces XML and why we choose it as our syntax. Section 5 describes the markup (tags) used in the documents and how they are used in automatic user interface generation. Sections 7–8 give examples of the software usage in example applications and show document markup examples along with their related user interfaces. Section 9 describes continuing work, and, finally, Section 10 summarizes and concludes.

2 Component Frameworks and the Document-centric Approach

Component- and object-based middleware platforms for large-scale heterogeneous environments are appearing at the fore of distributed systems research. Such systems provide support for object instantiation, discovery, naming, and communication based upon remote method invocation mechanisms (c.f., CORBA [9] or Enterprise Java Beans [10]). Continued evolution of such systems provides for a wide-ranging set of extensions to the basic model. Example directions include supporting 1) availability, fault-tolerance, and persistence to object stores [11], 2) uniform, language-independent "wrapping" around arbitrary data producers and consumers to make them look like objects to the system [12], 3) group communication primitives (i.e., tuple-spaces [13], reliable multicast [14], queued event notification [15]), and 4) system facilities for *dynamic, ad hoc* creation or extension of *end-user applications* from a set of constituent distributed components.

Designing systems with such a plethora of components is difficult even if it is simply considered a massive integration effort (i.e., even though many of the constituents exist); the more interesting and challenging design issue is that of determining the syntax and semantics used to describe components and how they are used. This defines the programming model. A traditional approach is to expose distributed components as objects that can be manipulated naturally from within a object-oriented programming language, albeit in a restricted fashion if the remote object isn't native. Actions are applied to data residing inside running applications using only well-defined library APIs to manipulate system state. On the other hand, a different approach is to externalize a portion of the system state in the form of *documents* that describe components and interactions. This allows state to be manipulated by *editing documents* in addition to allowing for access via an API — basically, *allowing system manipulation via authoring in addition to programming.* We call this relatively novel second approach a *document-centric* approach, and will focus on it for the rest of this paper.

Any Turing-complete language paired with a network can be used to encode the specification of a program built from distributed components — regardless of whether it is object-oriented, imperative, functional, or whatever. Thus, a "document-centric" model does not provide additional functionality (at least not fundamentally), but instead only adds an additional layer of indirection. Documents are written using a declarative style and used in addition to application code. Their function is to focus on the specification of data and elide (or greatly reduce the amount of) control flow information. Using a document-centric framework to expose this control/data separation (i.e., where documents are first-class entities rather than productions) provides a number of advantages, including

- aligning with the needs of cross-enterprise data-sharing, where the right thing to standardize is documents rather than APIs [16]
- simplifying inter-object usage specification, separating it from the programming chore and/or use of a particular monolithic application (manipulating documents to affect change is often simpler than editing and rebuilding programs, and less opaque than figuring out how to do through an non-standardized application user interfaces)
- providing for syntactic fault-tolerance ala HTML
- allowing clean incorporation of metadata, which can simply be directly added to a document rather than added as a new method or instance variable in an object
- providing for application-independent storage and manipulation — e.g., to maintain consistent per-object preference propagation, or to provide a single, concrete point for updates to a user's environment

This disassociation of programs/UIs from the objects they reference is similar to the Model/View/Controller architecture from Smalltalk [17]. In the M/V/C architecture, data (the model) is separated from the presentation of the data (the view) and events that manipulate the data (the controller). Similarly, documents in our system act as the glue that associates data to user interfaces/programs that manipulate and view that data. This strict delineation 1) provides client device independence, 2) provides program/UI language independence, 3) exposes program/UI to referent objects mappings: they become explicitly manipulable, 4) makes explicit what objects to manipulate and how they can be manipulated, and 5) can be used to generate user interfaces when custom ones are not available or unacceptable.

This document-centric distributed object management framework is illustrated in Figure 1.

This project investigates using a document-centric framework for specifying interaction with and between a distributed set of "services" available over a wide-area internetwork[1], for example, the set of services made available in a series of "smart spaces." It focuses on two aspects: description of the available services and flexible association of user interfaces to these services. We limit the scope of the discussion to a single (varying) collection of objects being referenced by a single (varying) user interface. This limits the problem domain so that we do not have to specify how the markup indicates paths of object-to-object interaction. Instead, we only have to describe interactions between the "endpoints:" users manipulating widgets to interact with a set of independent services. The focus, then, is on these two endpoints.

We proceed from the following observations:

- Services require concrete, immutable interface descriptions.
- Clients (or system proxies) need to manipulate and save collections of these interfaces.
- We would like a single document format that provides both functions.

From these observations, we establish our design. A single document format is shared among all entities in the system. At the "server-side," documents act as static interface definitions, similar to the use of IDL in CORBA. Elsewhere, documents act as a stable

[1] We call application components that use our framework "services" to contrast with the more generic term "objects."

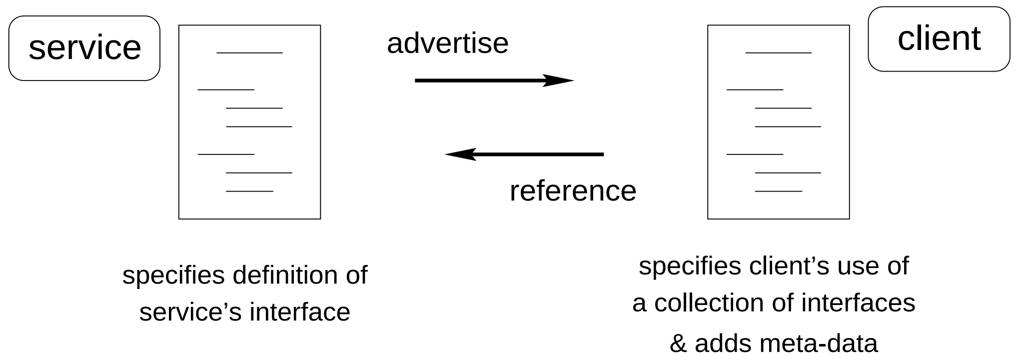


Figure 1: The Document-centric Model: Services are described by immutable, static documents that advertise the definition of their interface (ala the CORBA IDL or a Java interface); Clients maintain documents that indicate how the a collection of interfaces are used and maintain metadata about the collection

but manipulable (composable/decomposable) format for specifying object collections and references, defining interactions between objects in a collection, defining the object interfaces expected by programs and user interfaces, and storing arbitrary metadata about referents.

Specifically, there are two challenges. The first challenge is defining a single document schema that:

- notates services' available functionality, or *interface*,
- associates relevant programs and UIs to collections of services, or, vice-versa, lists the service interfaces expected by particular programs
- can flexibly compose and decompose based on constituent elements, and
- allows for easy incorporation of service-specific meta-data (i.e., without affecting existing functionality that does not expect it and in a self-describing manner).

The second challenge is providing software that can use documents written in the schema to generate user interfaces when custom ones are not available, mixes custom and generated user interfaces as necessary, and implements the run-time environment.

3 Project Approach

Our approach to the problem is threefold. Leveraging the eXtensible Markup Language (XML) [18] for syntax, we develop our schema as an XML document type definition (DTD). The schema provides markup tags for language-independent service descriptions and for mapping UIs (programs) to referent objects and vice-versa. We then build software that can heuristically generate UIs from these service descriptions without associated custom UIs, and allows mix-and-match use of custom and generated UIs. Additionally, we built an index application that lists the collection of available UIs and objects, allowing a combination of them to be interactively selected for presentation on the user's machine. Finally, we prototype applications that use the model, manually construction and editing documents to simulate how programs would automatically manipulate them.

For our prototype application domain, we focus on a set of location-based services [7] that provide software remote control of room devices from a wirelessly-connected laptop computer. Manipulations of the applications' documents allows the controls to adapt as the environment changes around the user. Specifically, the manipulations provide for

- the remapping of a portion of an existing user interface to a new room control (for example, due to movement of the terminal)
- viewing of arbitrary subsets and combinations of the functionality available in the space, and
- mixing generated user interfaces with custom user interfaces

This functionality is easily represented as operations on documents containing the associations between between programs/UIs from the objects they reference, exactly the model as described above.

4 XML

We have chosen to build atop the extensible markup language (XML) for our schema design, leveraging its allowances for the creation of custom, application-specific markup languages.

XML is an SGML subset providing self-describing custom markup in the form of hierarchical named-values and advanced facilities for referencing other documents (ala the HTML <href> tag). It is one protocol among a group that is touted as the successors to HTML (The companion protocols are XSL for style sheets and XLL for new linking mechanisms). XML includes ability to specify, discover, and combine a group of associated document schemata — otherwise known as *document type definitions* or DTDs. Examples include a growing set of metadata markup proposals such as Resource Description Format, the Dublin Core, XML-Data, and others.

Unlike HTML, the set of tags in XML in flexible; the tag semantics are defined by a document's associated DTDs. A key property of XML, then, is that it is *dependent* on these schema to be useful, and dependent on agreements in schema to allow interoperability. Thus, the problem is defining schema syntax (the tagset and their relationship) and agreeing on how a schema's associated "browsers" (borrowing the HTML term) semantically interpret these tags.

We believe there is a natural synergy between XML's need for schemata and the specification requirements of distributed object systems — the former provides a self-describing and extensible syntax with a rapidly expanding set of metadata tags; the latter provides a programming model for “web objects” described in XML.

5 XML Tags for Object Description

We use six tags in our initial design. Other tags that appear in our documents are assumed to be application-specific metadata, and can be ignored by programs that do not understand them. We now describe each tag in turn. The actual DTD specification is provided in the Appendix A.

The <object> tag is a container tag. It has one optional attribute, “name”, which is either a string or reference identifying the type/class of the object interface being described. It can contain at most a single <label> tag, zero or one <addrspec> tags, any number of <ui> tags, and any number of <method> tags. When converted to a user interface, an <object> is instantiated as a frame (a container for other widgets).

The <label> tag provides a text description of the object which contains it. It has no optional attributes. It can contain no additional internal tags except those providing text formatting. When converted to a user interface, the <label> tag is used as a title for its parent object’s frame.

The <addrspec> (address specification) tag indicates the address and port number on which its parent object listens for method invocations and events. An uninstantiated object will have no addrspec tag. It can contain no additional internal tags and does not have any optional attributes. When converted to a user interface, the <addrspec> tag is used as the location to which any method calls are sent (currently via string-based UDP messages).

The <method> tag defines the name of a method that can be invoked on the object in which it is contained. It has two optional attributes: “name”, which is name of the method call, and “lexType”, which indicated the lexical type of messages returned due to the method call (the list of lexical types is described below). The <method> tag can include (only) zero or more <param> tags. When used in automatic user interface generation, each <method> tag is mapped to a frame with contents. The name of the method is placed on a button at the top of this frame; pressing this button invokes the method call. At the bottom of the method frame, a label is appended for textual representation of replies. Note that in our system, method invocations and returns are asynchronous, event-based messages (like active messages [19]) rather than blocking remote procedure calls. Thus, update events (“replies”) can actually occur at any time, independent of the manual invocations at the client. In this manner, <method> tags can also be used as a means for subscribing to updates from the object.

The <param> tag indicates a parameter to the <method> tag that encloses it. It has two optional attributes, “lexType”, indicating the lexical type of the parameter, and “optional”, a boolean tag that indicates whether the parameter is required or optional. The <param> tag may have no additional internal tags, and its contents are assumed to be the name of the parameter. For UI generation, parameters are mapped to individual user interface widget objects. Each of these widget objects support a get_val method that returns the current widget setting. It is used to marshall the parameters for method invocations. The mapping from lexical type to UI widgets is described in Section 6.

The <ui> tag is used to associate a particular program to the object in which it is specified. The contents of the tag is assumed either to be a string indicating the name of an existing user interface object that will reference the document object description (assumed to be known or discoverable out-of-band), or the address and port number where such a user interface object can be requested. Only the former is currently implemented. It has one possible attribute, “lang”, indicating the language of the indicated program.

In our framework, documents are used in addition to application code, not instead of it. The documents act to specify what programs are needed and how they are run (via the use of <ui> tags at various levels in a hierarchical description of a service). The assumption is that the indicated applications will reference the documents, respecting the indirection exposed by the document.

6 Automatic User Interface Generation

Many of the mechanics of generating user interfaces from interface descriptions were described in the preceeding section. The remaining features to be discussed are the heuristic mapping from lexical types to user interface widgets, and how custom user interfaces indicated by a <ui> tag can be intermingled with these custom user interfaces.

We currently have implemented mappings from lexical types to objects wrapped around Tk [20] UI widgets in the mash shell [3]. Permissable lexical types include int, real, boolean, enum, string. The int and real type can have an optional range modifier. They are mapped to widgets as follows: an int or real with a “range” modifier is mapped to a scale widget (a slider). Without a range modifier, they are mapped to an entry widget (a type-in box). A boolean is mapped to a check-button widget (a toggle switch). An enum is mapped to a list of radio-buttons (one-of-N list selection). A string is mapped to an entry widget.

As for co-mingling these generated collections and existing UIs referenced in <ui> tags, the granularity of reference is at the level of individual objects. Thus, all objects receive a frame, and it is filled with either the custom-generated contents mapped from <method> and <param> tags, or the existing UI. The latter is handed a window handle and is expected to instantiate itself as a child of that window handle.

7 The Framework in Action: Examples

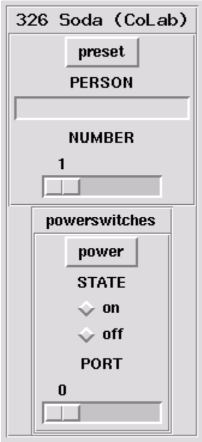
We now illustrate some examples. Each highlights a different element of the design of the overall architecture.

The first example shows an XML document that describes the interface to a portion of the functionality available in Soda Hall’s “CoLab” (Collaboration Laboratory, borrowing Xerox PARC’s terminology) and the resulting automatically generated user interface to it, as shown in Figure 2. The document describes two objects, one contained in the other. The outer object implements a method for setting a preset for the entire room; the inner object is one of the objects referenced by the outer object (i.e., one of the things set by the preset) and implements its own interface independently — an interface to a pair of power switches in the room. The <param> tags contain various lexical types, illustrating our use of heuristic mapping to widgets. This utility of this basic functionality is that it allows users the possibility to interact with dynamically discovered objects, not just programmatically, but also directly.

The second example illustrates the combination of a custom user interface with a generated one. The document is identical to that in the previous example except a single new tag is added: a <ui> tag to the internal (power switch) object, as shown in Figure 3(a). This causes that object’s interface to be replaced by the UI object referenced in the tag rather than generated on-the-fly. The resulting difference is illustrated in Figure 3(b). This example illustrates how dynamic extensions to existing applications can be seamlessly

```
<object name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <object name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </object>
</object>
```

(a) XML document



(b) User interface

Figure 2: An example document and generated user interface.

incorporated using our architecture, a form of "plug-in" architecture similar to that used in, say, Photoshop or with Visual Basic extensions.

The third example illustrates use of the indirection exposed by the use of the "document-centric" model by replacing a referent under a multi-object <ui> tag. The document fragment shown in Figure 4(a) is assumed to be used by an existing application. The user interface for this application is a custom-designed monolithic interface referenced in the topmost <ui> tag. In Figure 4(b), one of the component objects in the container object has been replaced. Because the type of the referenced object remains the same, only the <addrspec> tag changes. The result of this change is that the application looks the same, but a portion of it now references a new service. This function illustrates the possibility for remapping interfaces due to, e.g., terminal mobility or fault tolerance. Specifically, the example takes a portion of the document describing the interface to the 405 Soda Hall seminar room and remaps the light switch portion of the interface to a new switch.

The fourth and final example illustrates the ability to use a subset of presented functionality. The document in Figure 5(a) is the same as that from Figure 4, except all the internal objects referenced from the outermost container object have been ripped out. The resulting user interface is presented in Figure 5(b). This example shows how a user can easily elide material not considered relevant or not frequently used. In this case, we leave only the interface to the light switch exposed, simulating the case where the user has chosen to save screen real estate because, e.g., controlling only the lights is the most common usage.

8 Indices and the End-user Environment

In addition to building software to parse the particular XML DTD and spit out interfaces, we need to provide the user with a way to manage the set of documents and available interfaces. We provide this functionality through use an "index" application, shown in Figure 6. One the left side of the application, all objects are listed by type and address specification. Each type has an associated document and an associated user interface. When one of the checkbuttons beside an object name is set, the associated user interface is displayed for use by the user.

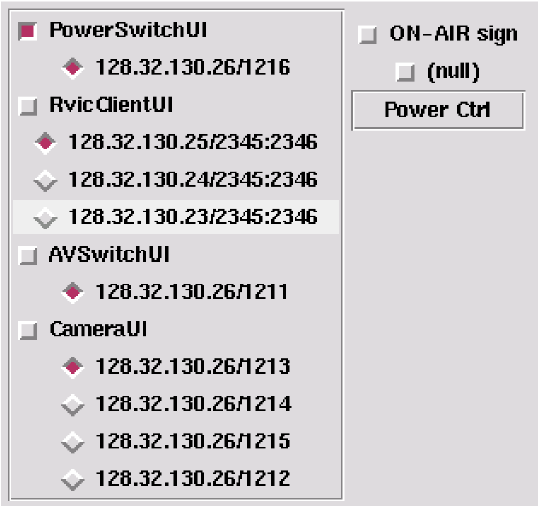


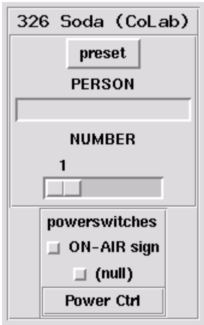
Figure 6: The Index application lists all available interfaces and allow the user to interactively select which ones he or she wishes to use. Illustrated here, the user has selected the user interface to the power switches in the Berkeley CoLab.

9 Continuing Work

Another set of applications that would be useful for the end-user are those that would allow easily manipulation of documents themselves rather than simply the results of having documents. Design and implementation of this piece is ongoing.

```
<object name="326">
        <label> Soda (CoLab) </label>
        <addrspec>spade.cs.berkeley.edu/0001</addrspec>
        <method name='preset'>
                <param lextype="string"> person </param>
                <param lextype="int:range=1-8"> number </param>
        </method>
        <object name='powerswitches'>
                <label>powerswitches</label>
                <addrspec>spade.cs.berkeley.edu/0002</addrspec>
                <ui lang=mash> PowerSwitchUI </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off"> state </param>
                        <param lextype="int:range=0-1"> port </param>
                </method>
        </object>
</object>
```

(a) XML document



(b) User interface

Figure 3: An example document and associated user interface, this time where a <ui> tag allows for the incorporation of a custom UI in addition to the generated components.

Currently, the <ui> tag can only reference local objects. We are in the process of implementing the remote retrieval protocol.

A logical next step of this work is dealing with mismatched types. For example, assume a light switch in some locale implements a different interface than the one in the user's home environment. Rather than require the use of a dynamically-generated user interface, we'd prefer to allow for the use of an existing user-interface. To do so, we must transparently remap method invocations to the new location and also remap the call parameters to match the new type. Incorporating such functionality allows far more flexibility in the reuse of existing user interfaces and intermingling of existing interfaces and discovered objects. The price is that it requires the use of external transformational operators that provide type coercion for method calls. Fortunately, such transformational operators could be written once, reused, and shared among the community of users; additionally, they could be chained together in order to provide new type-to-type coercions. This functionality is a natural extension of our framework. The difficulty of this approach is not in creating these mapping operators and storing them in a shared repository, but instead that of building the use of them into the end-user software. Users should be able to visually manipulate object mappings and the correct transformations should be done automatically. As a concrete example, this means that when a new light switch is discovered, the user should be able to indicate which program element should manipulate it, and any required remapping of method calls — i.e., document manipulations — should be done automatically, though possibly heuristically. Additionally, users should then be able to easily modify these mappings. Currently, this is all done via manual manipulation of documents rather than automatically.

Another important extension of this work is designing how to notate one object's use of other objects so as to allow for, e.g., multiple interfaces. The transformational operators described above are simple examples of such objects, in that they require separation of input and output interface descriptions. This requires extension or modification of our schema.

Finally, in order to allow for programmers to more easily use this document-centric model — without having to manually create interface description documents — we would like to automatically generate the documents from Java objects and other distributed component system pieces. To do so, we can leverage the CORBA Interface Definition Language (IDL) [9], for which there are mappings to C, Java, Ada, and other languages. We can integrate components written in these languages by creating a mapping from IDL descriptions to our XML schema and implementing it. Our object interface schema will need to be extended to support structured types in order to allow such a mapping.

10 Summary

We have described a document-centric framework for description and interaction with entities in a distributed object system. We have shown how the framework allows for:

- the remapping of a portion of an existing user interface to a new room control (for example, due to movement of the terminal)
- viewing of arbitrary subsets and combinations of the functionality available in a "plug-in"-style architecture, and
- mixing dynamically-generated user interfaces with existing user interfaces.

The use of a document-centric framework in addition to a conventional object-oriented programming language

- provides client device independence,
- provides program/UI language independence,
- exposes program/UI to referent objects mappings: they become explicitly manipulable,
- makes explicit what objects to manipulate and how they can be manipulated, and
- can be used to generate user interfaces when custom ones are not available or unacceptable.

```
<object name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <object name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/6902</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </object>
        <object name='vcr'>
                ...
        </object>
        ...
</object>
```

(a) Original XML document

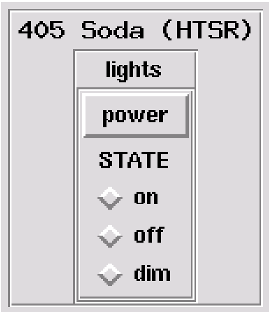
```
<object name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <ui lang='tcl/tk'>htsr.cs.berkeley.edu/6903</ui>
        <object name='lights'>
                <label>lights</label>
                <addrspec> 205south.sims.berkeley.edu/9999 </addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </object>
        <object name='vcr'>
                ...
        </object>
        ...
</object>
```

(b) Document with replaced referent

Figure 4: Remapping of function by replacing a referent under a multi-object <ui> tag. A fragment of the "original" document is show in (a); the modified document is shown in (b), where the only difference is the new <addrspec> tag. (The <addrspec> tags are highlighted.)

```
<object name="405">
        <label> 405 Soda (HTSR) </label>
        <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
        <object name='lights'>
                <label>lights</label>
                <addrspec>htsr.cs.berkeley.edu/0000</addrspec>
                <method name='power'>
                        <param lextype="enum:on,off,dim"> state </param>
                </method>
        </object>
</object>
```

(a) XML document



(b) User interface

Figure 5: Subsetting functionality. The example illustrates how functionality can be aggregated or subsetted by modifying the document associated with a program. The full description of the interface to 405 Soda has been cut down so that only a single object remains. The user interface is updated accordingly.

To implement our scheme, we designed a XML schema and accompanying software that

- notates services' available functionality, or *interface*,
- associates relevant programs and UIs to collections of services, or, vice-versa, lists the service interfaces expected by particular programs
- can flexibly compose and decompose based on constituent elements, and
- allows for easy incorporation of service-specific meta-data (i.e., without affecting existing functionality that does not expect it) via the self-describing nature of XML.

Using a peer document or "description" alongside an application to provide for much of the flexibility described here has been described before [21], but only in the abstract. It is the advent and growing popularity of XML and distributed object programming that synergistically combine to give us a syntax for these descriptions and a concrete framework for their use.

A Schema DTD

The document type definition for the XML files used by our system is as follows:

```
<!ELEMENT object (label?, addrspec?, ui*,
                  method*, object*)>
<!ATTLIST object
   name  CDATA   #REQUIRED>
<!ELEMENT method (param*)>
<!ATTLIST method
   name  CDATA   #REQUIRED>
<!ELEMENT param (#PCDATA)>
<!ATTLIST param
   name  CDATA   #REQUIRED
   lexType  (int | real | boolean | enum
                 | string | ...) 'string'
   optional  #BOOLEAN>
<!ELEMENT label (#PCDATA)>
<!ELEMENT addrspec (#PCDATA)>
<!ELEMENT ui (#PCDATA)>
```

References

[1] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, pages 80–86, January 1997.

[2] A. Fox, E. Brewer, S. Gribble, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Transcoding. *ASPLOS*, 1996.

[3] Steven McCanne, Eric Brewer, Randy Katz, Lawrence Rowe, Elan Amir, Yatin Chawathe, Alan Coopersmith, Ketan Mayer-Patel, Suchitra Raman, Angela Schuett, David Simpson, Andrew Swan, Teck-Lee Tung, David Wu, and Brian Smith. Toward a Common Infrastructure for Multimedia-Networking Middleware. *Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[4] B. MacIntyre and S. Feiner. Future Multimedia User Interfaces. *Multimedia Systems Journal*, 4(5):250–268, October 1996.

[5] Brewer, Katz, et. al. *A Network Architecture for Heterogeneous Mobile Computing*. submitted for publication, IEEE Personal Communications.

[6] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communication of the ACM*, 36(7), July 1993.

[7] Todd Hodes, Randy Katz, E. Servan-Schreiber, and Larry Rowe. Composable Ad hoc Mobile Services for Universal Interaction. *Proceedings of the 3rd ACM International Conference on Mobile Computing and Networking*, pages 1–12, 1997.

[8] David Krieger and Richard Adler. The Emergence of Distributed Component Platforms. *IEEE Computer Magazine*, pages 43–53, March 1998.

[9] Object Management Group. Common Object Request Broker Architecture. http://www.omg.org/.

[10] Sun Microsystems. Enterprise Java Beans. http://java.sun.com/ejb.

[11] J. Eliot, B. Moss, and Tony L. Hosking. Approaches to Adding Persistence to Java. *First International Workshop on Persistence and Java*, September 1996.

[12] Charles Axel Allen. Automating the Web with WIDL. *World Wide Web Journal*, 2, 1997.

[13] IBM Almaden Research. TSpaces. http://www.almaden.ibm.com/cs/-TSpaces.

[14] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *ACM SIGCOMM 95*, pages 342–356, August 1995.

[15] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and M. Frans Kaashoek. Rover: A Toolkit for Mobile Information Access. *Proceedings of the Fifteenth Symposium on Operating System Principles*, December 1995.

[16] Robert Glushko. The XML Revolution. UC Berkeley SIMS Symposium Presentation, April 1998.

[17] G. Krasner and S. T. Pope. A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, August/September 1988.

[18] World Wide Web Consortium. eXtensible Markup Language. http://w3c.org/XML/.

[19] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. *International Symposium on Computer Architecture*, May 1992.

[20] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[21] T. Hodes and R. Katz. Composable Ad hoc Location-based Services for Heterogeneous Mobile Clients. *ACM Wireless Networks*, 1998. Special issue on Mobile Computing, to appear.