UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

UNDERSTANDING UI INTEGRATION: A SURVEY OF PROBLEMS, TECHNOLOGIES, AND OPPORTUNITIES

Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati,
Maristella Matera and Regis Saint-Paul

October 2006

Technical Report # DIT-06-064

Understanding UI Integration: A survey of problems, technologies, and opportunities

Florian Daniel[1], Jin Yu[2], Boualem Benatallah[2], Fabio Casati[3], Maristella Matera[1] and Regis Saint-Paul[2]

[1] Politecnico di Milano, Italy
{daniel,matera}@elet.polimi.it

[2] University of New South Wales, Sydney, Australia
{jyu,boualem,regiss}@cse.unsw.edu.au

[3] HP Laboratories, Palo Alto, CA, USA
fabio.casati@hp.com

1. Introduction

The problem of facilitating the creation of applications from components has been one of the biggest areas of investigation in software engineering and data management over the past 30 years. It has led to a large body of research and development in such areas as component-based software engineering, middleware, and service composition. While results from these efforts simplify integration at the data or application level, little work has been done to facilitate integration at the presentation level. Everybody who has developed graphical applications is aware that the development of user interfaces (UIs) is one of the most time-consuming parts of application development, testing, and maintenance [1]. This would suggest that reuse is essential also in UI. However, while UI development today is facilitated by frameworks (such as Java Swing) providing pre-packaged classes with UI functionality such as buttons, menus, and the likes, the integration of coarse-grained and possibly stand-alone applications at the UI level has received little attention.

In this work we investigate the problem of graphical UIs (GUIs) integration; that is, integration of components by combining their presentation front-ends, rather than their application logic or data. The granularity of components is that of stand-alone modules or applications, and the goal is to build composite applications that leverage the components' individual UIs to produce richer, composite UI applications. The need for such integration is manifest, and examples are numerous: applications overlaying real estate information over Google maps, aggregated dashboards showing consoles monitoring different aspects of a computer's performance [16], or "web" operating systems that allow coordinated interactions with multiple applications on the same web page [17]. All these examples require coordination among application UIs (e.g., zooming out on a map means that overlaid information on houses for sale must change as well).

The objective of this paper is to identify the basic characteristics of UI integration as a research discipline, discuss its main issues, and present the different approaches that can be taken to address them. Specifically, we describe and exemplify the characteristics, challenges and opportunities of UI integration in comparison with the two main other kinds of integration, namely data and application integration. This is important not only to understand why UI integration differs from other integration problems and hence why it requires unique technologies and methodologies, but also to understand the similarities, as there are many lessons that can be learned from these other types of integration. Then, we characterize the main dimensions of the UI integration problem, which gives us a framework under which to analyze existing approaches to UI integration and discuss their characteristics. Finally, we discuss what is missing in current approaches and how we believe that the field should or will evolve to facilitate UI integration.

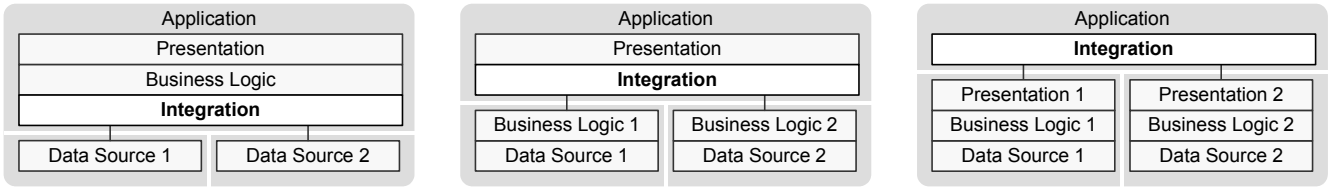
2. The different layers of integration

To describe the different types of integration we use a simple but concrete scenario, based on actual applications developed within Hewlett-Packard. Consider a set of applications that monitor the performance and quality of systems, networks, services, and business processes. For example, a *system monitoring* tool logs system metrics

(such as CPU utilization) for a set of machines and sends alerts in case they go above certain thresholds, while a *process monitoring* application looks at business process executions and report on key performance indicators such as process duration or rate of process instantiation. Each of these applications, like most modern applications, is structured into three layers: presentation, application (also called *business logic* layer), and data.

While monitoring applications have been developed independently over time, there is an increasing need to look at them in an integrated fashion. This is useful to perform *root cause analysis* (understand what system problem is the cause of a delay at the process level) as well as *business impact analysis* (understand what is the "damage", at the process level, of a failure or performance degradation in a system or service), and in general to have an end to end view of the managed IT systems. As a simple integration example, assume that our integrated monitoring application is formed by two components: a business process monitoring tool and a system monitoring tool. A desired behavior of the integrated application is that when the user selects a specific process to be visualized in the process monitoring tool, then on the system monitoring window the different systems supporting the process (and their health and availability status) are displayed. We next examine the different types of integration that can be used to achieve this purpose.

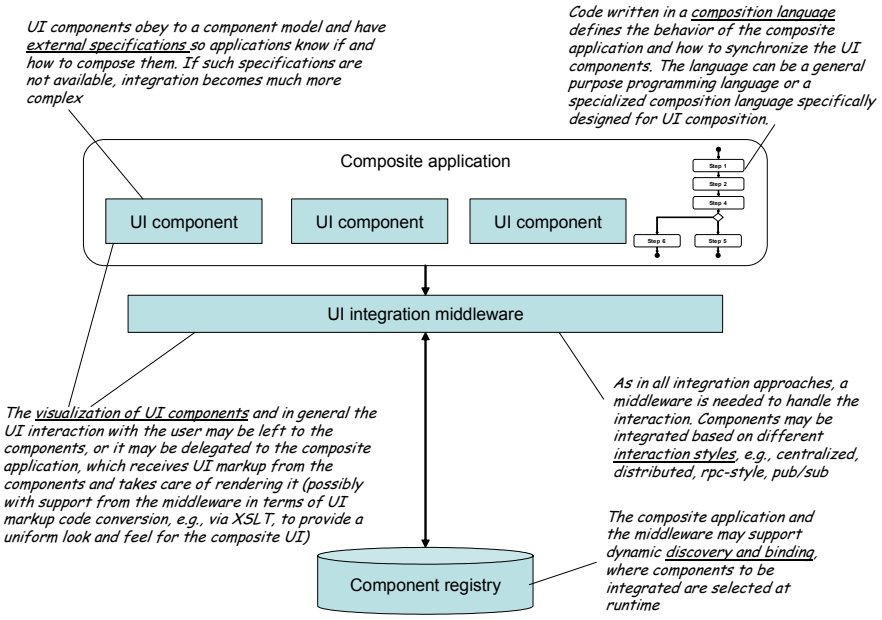
In **data integration** approaches, composite applications have their own presentation and application layer, while the data layer is in fact an integration of the data independently maintained by the component applications (Figure 1(a)). In the management scenario, the different monitoring applications collect data in their local repositories, unaware of being the object of integration. An integration layer brings data together and exposes a unified, homogeneous view to the composite application. The integration layer can be materialized or it can remain virtual.



(a) Integration of different data sources

(b) Integration of distributed business logic elements

(c) Integration of two autonomous applications



(d) A possible UI composition framework. The registry may or may not be present.

Figure 1 Component integration at different levels.

Data integration presents a number of problems, ranging from how to understand and resolve the semantic mismatch between the component data models (e.g., the component models may use the same term with different meanings) to how to construct and maintain virtual schemas and map queries and results from virtual (aggregated) to component schemas. Data integration is often used because it requires little "cooperation" from component applications. One can always tap into the applications' databases by means of SQL queries, or using Enterprise Information Integration (EII) technologies [18]. The drawback is that it requires a significant effort to understand the data models, to analyze semantic heterogeneities, and to maintain the composite schema in the wake of changes to the component data models [18].

In **application integration**, a composite application would have its own UI, but its business logic layer is, at least in part, developed by integrating functions exposed by the component applications (Figure 1(b)).

In the management scenario, for example, the monitoring applications could expose an API that allows clients to retrieve performance data for a certain system, or to subscribe to alerts for performance degradations. The composite application would use this API to get information, correlate it across the different monitoring applications, and display consolidated end-to-end monitoring information on its GUI.

When possible, i.e., when such an API is available, this integration model has many benefits: 1) the granularity of functions provided by the component applications is generally well suited for high level integration (for example, we can tell a monitoring application to begin monitoring machine XYZ, without considering the detail of how this affects the data in the integrated application's database), and 2) it is more stable as the component application is aware of the integration (it is exposing the API), and hence will drive towards making the interface more stable across versions.

Application integration has been thoroughly studied over the last thirty years, giving rise to technologies such as RPC, object brokers (such as CORBA), workflows, and Web services [19]. Research in this field has identified several problems, only partially solved by the above technologies. There are in particular four classes of problems that have been studied and that are also key issues in UI integration:

1. Define a model and language to specify components
2. Define a model and language to specify how to perform the composition
3. Define a way to discover and bind to components, possibly at runtime
4. Provide an approach to support the interaction and communication among the components.

Ideally, models and specifications have to be simple enough to be understood and easily adopted by users, formal enough to be parsed by applications or tools and provide an added value (e.g., search for components, analysis of compatibility between components, etc), and complete enough to model a wide range of concerns.

Finally, **UI integration** composes applications by reusing their own user interfaces. This means that the presentation layer of the composite application is itself composed, at least in part, by the presentation layers of the components (Figure 1(c)). UI integration, is particularly applicable in cases where application or data integration is not feasible (e.g., the applications do not expose a business level API), or where the development of a new UI from scratch is too costly (e.g. the component application often change or its UI is complex). While UI integration can be extremely beneficial, the lack of an "UI middleware" can make this task hard to perform. We discuss UI integration in detail in the next sections.

3. Dimensions of the UI integration problem

We now discuss the different dimensions of the UI composition problem. Specifically, we discuss the four dimensions that are common with application integration (although reinterpreted in the context of UI integration) and discuss a fifth dimension that is related to how aggregated UI information is visualized (see Figure 1(d)). We then use this framework to analyze current UI technologies.

3.1 Component model and external specifications

This dimension deals with the characteristics of the UI component as presented to the integrating developer or application.

In application integration, components are essentially characterized by an API and possibly by a component model (e.g., the CORBA component model). In data integration components are described by database schemas or XML schemas. In UI integration, only recently have external specifications become a topic of interest. Indeed, integration in UI was mostly intended as reusing class libraries to facilitate development. In the management example, the problem here is how the different monitoring components can be accessed and what is shown on their UI can be modified so to achieve a coordinated, integrated display.

We distinguish among the following "degrees of interoperability" allowed by a UI component's interface:

GUI-only: This is analogous to a traditional monolithic desktop application. All interaction with the component is performed via the component's UI. The only way to integrate a component application is to have intimate knowledge of its UI, be able to track the mouse position and/or to intercept the text entered by the user, and in this way understand what is shown by the component's UI and possibly even execute actions that cause UI modifications (e.g. by having the composite application simulate mouse clicks or keyboard strokes). Integration in this case is a daunting task.

Hidden interface: The component has an interface that allows controlling its UI, but it is not publicly described. This is the case of many Web applications. For example, if we are integrating a Web application, we can control the content by sending HTTP requests and displaying the response. There is no need to control the component application by simulating mouse moves. However, the interface is not guaranteed to be stable, as there is no commitment to its stability by the application provider. Also, it is hard to detect UI changes initiated by the user directly on the application, as there is no support for the communication of UI events.

Published interface: The component provides a description of its UI and an API to manipulate the UI at runtime. The API can be at different levels of abstraction. A low level API may allow control of individual UI elements such as button or text areas. A high level API would instead expose a set of "entities" (e.g. "system" or "network" in our management example), as well as operations to change the UI based on entities, such as "show status of system xyz".

3.2 Composition language

This dimension refers to the programming model and language through which developers can integrate components to create the composite application. In the management example, the problem is how to specify the composite application.

In data integration, composition is often done via SQL views that allow a global schema to be expressed as a set of views over local schemas [2]. In application integration, composition is done either via general-purpose programming languages such as Java, or via dedicated application integration languages, such as workflow or service composition languages (e.g., WS-BPEL [3]). Both data and application integration thus provide mature composition or integration languages. Although UI integration analogously will require similar technologies, so far only little work has been done in this direction.

In summary, we distinguish between the following two composition environments or languages:

General-purpose programming languages: The composition is performed by means of third generation languages like Java. Such languages are very flexible but lack abstractions for the composition of coarse-grained components (e.g., facilities for component discovery and binding, or high-level primitives for synchronizing what is displayed by UI components).

Specialized composition languages: These are a high-level languages (nowadays usually XML-based) tailored to the composition of UI components at the level of abstract/external descriptions. This idea is similar to service

composition languages. The main benefit of such languages is that of a higher level programming of the composition that leverages the characteristics of the component model. For example, if the component model has the notion of *entities* as described above, the composition language may provide primitives to inspect entities displayed by a component, for changing the entity being visualized, etc.

3.3 Component visualization

This dimension characterizes who is in charge of visualizing a UI component: the component or the composite application. In the management example, the issue is whether components display their own monitoring dashboard or whether the composite application receives UI markup code from the components and takes care of rendering it. UI markup languages can be categorized as document markup and UI markup languages. The former (e.g., (X)HTML, WML) describe document properties, the latter (e.g., XAML, XUL) describe application user interface properties. Markup visualization requires interpretation by a rendering engine (e.g., a browser), translating the descriptions into graphical elements. Markup specifications typically describe static UI properties, while dynamic behaviors are achieved by means of scripting languages (e.g., JavaScript).

In summary, we distinguish the following types of component visualization:

Component-rendered UI: The rendering and displaying of the UI is handled by the component. The composite application is a collection of the components' UIs. This is the case of classical desktop applications that leverage executable components with linked graphics libraries.

Markup-based UI: UI components may return UI code and delegate the actual rendering of the final UI to the composite application, or to the environment in which the composite application is executed. The composite application must thus be able to interpret the components' UI code, and must allocate suitable space on the display for the rendering of the components.

3.4 Communication style

This dimension deals with the topology of the interaction between components and the composite application. In the management example, the issue is how the monitoring components exchange UI events to notify user actions significant to the composite application (e.g., a user changes focus to a different system and therefore all UI components need to change focus as well) or to receive instructions on what to display.

In data integration, components typically are passive participants that do not initiate communications with the integrating application. This perspective is also common in application integration, where we have a centralized entity (the composite application) that invokes components as needed, although fully distributed interactions are becoming more common (e.g., where a seller, a buyer, and a shipper interact without a central coordinator).

In UI integration we can also distinguish between *Centrally-mediated communication,* where the composite application has a central coordinator that receives events from components and issues instructions - e.g., via API calls - to modify the components' UIs, and *direct component to component communication*, where the composite application is a cooperation of components, and there is no first-class application orchestrating the activities of the other components.

An additional distinction, orthogonal to the above one, is between *RPC-style* interaction (where information is exchanged via method calls and return data) and *publish-subscribe* interaction, where applications communicate in a loosely-coupled way via messages exchanged through message brokers, where messages are distributed based on the message content (or *topic*, as it is often called in the literature).

3.5 Discovery and binding

Binding is a key issue in any kind of integration. The problem here is how to identify which are the components to be integrated and how to obtain a *reference* to it (e.g., an object ID or an URI). This can typically be done statically (at design or deployment time) or dynamically (the reference is retrieved at runtime, after a *discovery* typically based on querying some registry). In the management example, the problem is how the composite application identifies and binds to the relevant monitoring applications.

In data integration, binding between different data sources typically occurs at design time, when the global data schema is defined. This is also the case for application integration. In fact, although technically the integration middleware allows dynamic discovery and binding [19,20], this flexibility is not often exploited because of the problems inherent in interacting with a newly discovered component, especially if provided by another company (e.g., the integration with the component has not been previously tested, it is not clear if the functional and non-functional properties are as advertised, etc..). In many cases applications resort to *hybrid* binding, where a set of potential components (e.g., a set or monitoring applications) is identified and tested at design time, and a subset of them is selected at runtime based on the needs at hand (e.g., based on the systems to be monitored). In hybrid binding, the discovery is static but the reference is obtained at runtime.

The same distinction is possible in UI integration, and this dimensions hence characterizes approaches based on whether they allow *static binding*, *dynamic binding*, or *hybrid binding*.

4. Composition technologies

In this section we review the most promising approaches to UI integration in the light of the dimensions presented above.

4.1 Desktop UI components

Historically, UI composition has first been seen for desktop applications. The introduction of component technologies eventually provided an environment that allowed applications developed using heterogeneous languages to interoperate. A typical example is given by ActiveX [8], which leverages Microsoft's COM technology for embedding complete application UI into host applications. Other examples are Apple OpenDoc or GNOME Bonobo. These technologies rely heavily on the operating system or on component middleware (e.g. CORBA) that provide for component interoperability.

By contrast, the Composite UI Application Block (CAB [10]) is a framework for UI composition in .NET. Its container service allows applications to be built upon loadable modules or plug-ins. Developers concentrate on reusable components that can be dynamically plugged into the container at runtime. CAB separates UI components from the business logic. CAB components can be used with any .NET language to build composite containers and to perform component-container communications. CAB further provides an event broker for many-to-many, loosely coupled inter-component communication based on a publish/subscribe runtime event model.

Eclipse's Rich Client Platform (RCP [9]) provides a similar framework. RCP includes an application shell (i.e. container) with UI facilities such as menus and toolbars. RCP offers a module-based API enabling developers to build applications on top of this shell. In addition, Eclipse allows UI components (i.e. Eclipse plug-ins) to be customized/extended via so-called extension points, a combination of Java interfaces and XML markups defining the a component's interface, which support the loose coupling of UI components.

In this category, *general-purpose programming languages* are typically used to integrate components (i.e. C# for CAB and Java for RCP), since the component interfaces are language-specific programming *APIs*. Components perform their *own UI rendering* and flexible communication styles are supported; i.e., *centrally-mediated* and *direct component to component*. Both *design-time* and *runtime* bindings are supported, the latter relying on language-specific reflection mechanisms.

Contrary to the ActiveX model, CAB or RCP are not OS-dependent, but rely on their respective runtime environments (e.g., JVM for RCP). The lack of technology-agnostic, declarative interfaces makes interoperation between technologies hard.

4.2 Browser plug-in components

Rich UI features in markup-based interfaces are often achieved, besides through JavaScript and dynamic HTML, by means of embedded UI components like Java applets, ActiveX controls, Macromedia Flash, or various media players.

The external interface of such components is very simple and usually only requires to set proper *configuration parameters* when embedding the components into the *markup code*, which represents the composition language. Plug-in components provide for their *own rendering*, and usually there is little further *communication* between components and the containing Web page, or among components. Component bindings are specified *at page-authoring time*; during runtime, the browser downloads and instantiates the components.

Embedded UI components are easy to use, but the lack of a systematic communication framework is a limitation. Communication with components can be achieved through ad-hoc JavaScript, but this is far from a uniform approach to deal with components. This limitation derives more from the browser's sand-box mechanism for the execution of plug-ins and is less ascribable to the component model itself.

4.3 Web mashups

Web mashups are websites that wrap and reuse contents provided by third parties as Web sites or services, often accompanied by a proper API. The first mashups couldn't rely APIs, as the actual content providers didn't know that their Web sites were wrapped into other applications. The first mashups with Google Maps, for example, predate the official release of Google's API for Maps [7]; the API is Google's answer to the growing number of hacked map integrations, where people read the whole AJAX code of the Maps application and derived the needed functionalities.

Publicly available APIs for mashups are still rare on the Web; mashups may thus still be derived from *hidden interfaces*. Integration is performed in an *ad-hoc* fashion by leveraging whatever programming language supported by the content source, on either client side (e.g., AJAX) or server side (e.g., PHP, Java, ASP.NET,…). Contents are typically provided as *markup code* and integrated in a *centrally mediated* fashion. The lack of infrastructure makes component-component communication difficult and only provides means to *statically bind* components.

Since component interfaces may not be stable, most effort in the development of mashups goes into manual testing. Due to the lack of framework support, code isolation is not guaranteed (i.e., code collision of two JavaScript source codes), and conflicts among UI components may occur. Building Web mashups remains a hard and time-consuming task.

4.4 Web portals and portlets

This approach explicitly distinguishes between UI components (the portlets) and composite applications (the portals) and is probably the most advanced approach to UI composition as of today (We use the vocabulary of Java portlets [3] but our considerations also hold for ASP.NET Web Parts [12]). Portlets are full-fledged, pluggable Web application components. Portlets generate document markup fragments (e.g. (X)HTML, or WML) that adhere to certain rules (e.g. common CSS styles) to facilitate content aggregation in portal servers to form composite documents. Portal servers typically allow users to customize composite pages (e.g., to rearrange or show/hide portlets) and provide single sign-on and role-based personalization. Typical portlets include weather reports, discussion forums, and stock quotes.

Analogous to Java servlets, portlets implement a *specific Java interface* as main abstraction of the *standard portlet API* (JSR-168 [3]), intended to enable developers to create portlets that can be plugged into any standard-conform portal server. JSR-168 defines a runtime environment for portlets (i.e., portlet container) and the Java API between container and portlet. Portal applications, in the case of Java portlets, are based on the *Java programming language*, while in the case of Web Parts, applications are programmed in *.NET*. The portal application aggregates the *markup outputs* of its portlets and manages the communications in a *centrally mediated* fashion. Portlets allow both *static* (i.e., design time) *and dynamic bindings*; during runtime, portlets could be made available in a registry, and users could be enabled to select and position them.

JSR-168 does not provide inter-portlet communication mechanisms, but work on this is underway (JSR-286 [5]). ASP.NET Web Parts support inter-part communication by means of shared data structures. However, shared data structures make Web Parts tightly-coupled; a publish/subscribe event mechanism might be more desirable.

Although portlets and Web Parts have similar goals and similar architectures, they are not interoperable. Web Services for Remote Portlets (WSRP [6]) addresses this issue at the protocol level: WSRP exposes remote portlets as Web services, and communications between portal server (WSRP consumer) and portlets (WSRP producer) occur via SOAP. Portal and portlets can thus be built with different languages and runtime frameworks. WSRP 1.0 does not support inter-portlet communication; ongoing work in WSRP 2.0 proposes an event distribution mechanism.

	UI component Model and external specification	*Composition language*	*Component visualization*	*Communication style*	*Discovery and Binding*
Desktop UI components	Published, programmable API	General-purpose programming language	Component-rendered	Centrally-mediated and component to component communications may be supported	Static and dynamic binding
Browser plug-in components	Published, basic interface (start-up configuration parameters)	Document markup code and JavaScript	Components are component-rendered	Centrally-mediated. Very limited inter-component communication via ad-hoc JavaScript	Static binding
Web mashups	Hidden interface; published API	General-purpose programming language (e.g., JavaScript)	Typically markup-based	Centrally-mediated	Static binding
Web portals and portlets	Standard interface based on public API; interface wrapped as Web service	General-purpose programming language (Java / .NET)	Markup-based	Centrally-mediated (inter-portlet communication under development)	Static and dynamic binding

Table 1 Comparison of current UI approaches.

5. Discussion

The analysis above emphasizes that many aspects common to integration in other areas and important also in UI integration are missing from the state of the practice. In particular, what is needed is a model and language to provide external specifications for UI components suitable for UI integration. Just as WSDL is seen as indispensable to Web services, we believe that a sound UI integration approach should be based on implementation-independent descriptions of the functional and non-functional properties that characterize a component. The specifications need to be abstract (high level) in the sense that they do not need to expose all the tiny details of the UI (if every detailed UI aspects is controlled by the composer, there is little benefit in reuse). In the management scenario, this means that integration is facilitated once the different UI components provide, for example, operations to change the display to focus on a specific process, service, or system.

The idea of portlets and WSRP goes into this direction, but their interoperability support is still poor. Mashups present promising examples, but lack any kind of agreed execution framework or standard. Browser plug-ins can be regarded as mature components, but again lack standardized interoperability mechanisms. In desktop component-based UI composition, interoperability is supported but is low level and restricts the composition to a given platform (e.g. Java or .NET) or operating system.

UI integration has also a need for synchronization among components as they all need to display related information. This implies that individual components must be able to publish events (again, these would be high level, application-specific events such as *changeInProcessDisplayed* as opposed to generic UI events such as

mouseClick). Component interoperability today is still heavily focused on component-composer communications or centrally mediated communications between components, but UI composition requires models and architectures to communicate UI events among components, possibly without a centralized mediation. In our management scenario, the *process monitoring* application and the *service monitoring* application would interact via events concerning related entities among the two applications, thus synchronizing the contents displayed by the autonomous components. Starting from standard component descriptions and interoperability solutions, specialized UI composition languages could then allow designers to compose UI components at a high level of abstraction and to disregard low-level implementation issues.

Coordinated interaction has been heavily studied in application integration but results have not percolated into models and languages for UI integration. This is partly because of the lack of adequate components (before components with external specifications are available it makes little sense to define language to compose them), but also because the problem is different and inherently difficult. In fact, in UI integration we need to mix P2P and distributed interaction style (where components communicate to synchronize the display) with procedural aspects, where for example a user action (e.g., user switching to a mode where only process-level information is displayed, but on all processes related to finance) generates the need for a sequence of steps to be executed by the composer (e.g., closing windows, changing focus on other, bringing new UI components into the display).

In summary, from the lessons learned in data and application integration we derive that platform-independent solutions, such as those leveraging standard Web technologies, are the most likely to succeed in the development of an open, flexible UI integration framework. As seen in Web services, we believe that a loosely coupled component model is best apt also to respond to the novel requirements and challenges that UI integration may pose. Finally, we note that runtime UI composition, when made available, will raise new and exciting challenges that go far beyond purely technical problems to include Human-Computer Interaction concerns such as usability, friendliness or esthetic considerations.

6. References

[1] Myers B. A., Rosson M. B. 1992. Survey on user interface programming. *SIGCHI'92.*

[2] Lenzerini M. 2002. Data integration: a theoretical perspective. *PODS '02*.

[3] G. Alonso et al. *Web Services: Concepts, Architectures, and Applications*. Springer, 2004.

[4] Abdelnur A.; Hepper S. 2003. Java Portlet Specification. <*http://jcp.org/en/jsr/detail?id=168*>

[5] Hepper S. 2005. JSR 286: Portlet Specification 2.0. <*http://jcp.org/en/jsr/detail?id=286*>

[6] OASIS 2006. Web Services for Remote Portlets. <*http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp*>

[7] Google 2006. Google Maps API. <*http://www.google.com/apis/maps/*>

[8] ActiveX Controls. <*http://msdn.microsoft.com/workshop/components/activex/activex_node_entry.asp*>

[9] Eclipse Rich Client Platform. <*http://wiki.eclipse.org/index.php/Rich_Client_Platform*>

[10] Smart Client. Composite UI Application Block. <*http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/cab.asp*>

[11] Merrill D. 2006. Mashups: The new breed of Web app. <*http://www-128.ibm.com/developerworks/library/x-mashups.html?ca=dgr-lnxw16MashupChallenges*>

[12] Walther S. Introducing the ASP.NET 2.0 Web Parts Framework. <*http://msdn.microsoft.com/asp.net/default.aspx?pull=/library/en-us/dnvs05/html/webparts.asp*>

[13] Apple OpenDoc. <*http://en.wikipedia.org/wiki/OpenDoc*>

[14] A. Halevy et al. Enterprise information integration: successes, challenges and controversies. *SIGMOD'05*.

[15] Schmidt D., Vinoski S. 2006. Object Interconnections. <*http://microsites.cmp.com/documents/s=9063/cujcexp2007vinoski/*>