Seamlessly Integrated, but Loosely Coupled – Building User Interfaces from Heterogeneous Components

Heiko Paulheim
SAP Research CEC Darmstadt
Bleichstrasse 8
64283 Darmstadt, Germany
heiko.paulheim@sap.com

ABSTRACT

User interface development is a time and resource consuming task. Thus, reusing existing UI components is a desirable approach for rapid UI development. To keep UIs maintainable, those components should be loosely coupled. Composing UIs of heterogeneous components developed with different technologies, on the other hand, is a non-trivial task not supported well by currently existing integration frameworks, and there is only little progress in automatizing the integration step.

In this paper, we introduce a framework for UI integration which is capable of handling heterogeneous UI components. It facilitates events annotated with RDF and ontologies for assembling user interfaces from loosely coupled components. With that framework, UIs can be composed semi-automatically, based on logic event processing rules.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User Interfaces*; D.2.13 [**Software Engineering**]: Reusable Software—*Reuse Models*

General Terms

Design, Algorithms

Keywords

User Interfaces, Integration, Component-based Software, Ontologies, RDF, Semantic Web

1. INTRODUCTION

Developing a software system's user interface is a time and resource consuming task which takes up to 50% of the system's total development time [8]. Therefore, reusing existing components is an important topic in UI development. While numerous technologies for developing highly appealing UI components, such as Flex, Silverlight, JavaFX etc., have been introduced during the past years, it has become more likely that reusing existing UI components involves the task of *integrating heterogeneous UI components*.

Although there are libraries for displaying e.g. Flex components within a Java application, such as *JFlashPlayer* or *DJ Native Integration* (see [13] for a comparison), those libraries do not yet provide a meaningful integration. To end up with a useful integrated UI, the integrated components need not only be displayed next to each other, but also react to actions performed with other components, such as highlighting related information in different components, or allowing the user to drag and drop objects from one component to the other. Only by allowing such interactions, the user will experience an integrated UI as being all of a piece. We use the term *seamless integration* to indicate such a type of integration.

Current UI integration approaches are still rather limited with respect to facilitating interaction between the integrated components [3], and the use of heterogeneous components makes things even worse. Furthermore, implementing *seamless integration* most often implies writing code specifically for the components that are supposed to interact with each other, as opposed to the paradigm of *loose coupling*, and leads to code-tangling and systems that are hard to maintain [4]. For heterogeneous user interface components, the glueing code may not only be scattered across components, but also be written in different programing languages, which further complicates maintenance.

In this paper, we introduce a framework for seamless UI integration based on RDF and ontologies [9]. The framework is based on an exchange of annotated events, which leads to a loosely coupled and well maintainable integrated system, and facilitates semantic event processing by using a centralized reasoning module.

2. BASIC FRAMEWORK ARCHITECTURE

Fig. 1 gives a high-level overview of our integration framework's architecture. Each UI component runs in a *container*, which is responsible for displaying the UI and provides different services for performing the integration with other components.

The *event handling* service is responsible for sending and receiving events to and from other components. To this end, all containers are connected via a common event bus. The *object transformation* service transforms objects to and from RDF for sending them along with events. The *drag and drop handling* service keeps track of objects dragged from and dropped to components and facilitates drag and drop even across heterogeneous components. The *components and state management* service manages the UI component's subcomponents, and their respective states.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

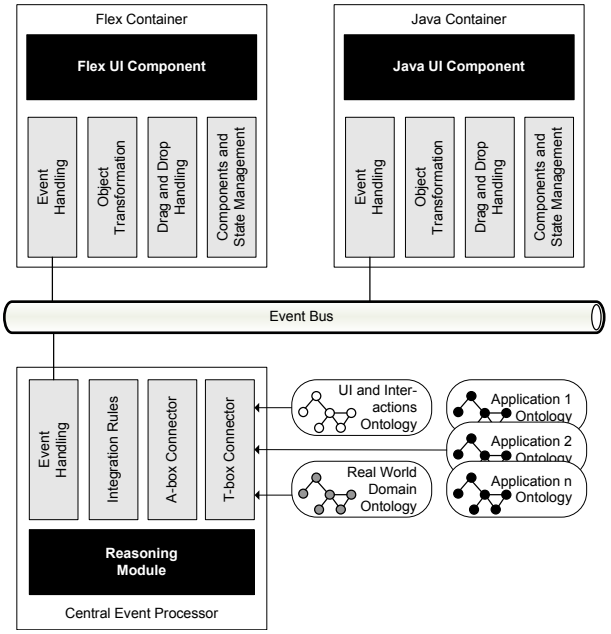

Figure 1: Architecture overview. Components run in containers and exchange annotated events, using a centralized, ontology-based event processor.

An essential component of our framework is the central event processor (CEP) and its reasoning module. The CEP is also connected to the framework's event bus by its own *event handling* service and can thereby send and receive events. It facilitates centralized event processing [3] by analyzing events and computing new events that are triggered by the ones that have been received. By incorporating information about the event's meta data as well as the objects contained therein, i.e. the semantics of an event, the reasoning module facilitates *semantic* event processing, as defined in [15]. The central event processor serves as an indirection to ensure loose coupling of the UI components.

To define the semantics of events, we use three different types of ontologies, formalized in F-Logic: An *ontology of the user interfaces and interactions domain* defines the basic concepts of UI components, as well as user and system actions. A *real world domain* ontology describes the real world entities the system deals with, such as banking accounts and customers or travel itineraries. Based on those two ontologies, a *specific application ontology* is defined for each component integrated in the system, defining the specific sub-components of that component, their behaviour, and the interactions they support [9].

These ontologies are provided to the reasoner by the *T-Box connector*, which reads them at system start time and loads them into the reasoner. Based on those ontologies, *integration rules* can be defined which control cross-component interaction. For example, a linking interaction is defined by the following integration rule: "When a select event is detected involving an information object, it triggers a highlight event for each other information object currently displayed which refers to the same real world object."

By using reasoning on a real world domain ontology, such integration rules may be formulated in a more general way. For example, a map component might declare that it can show the position of *everything that has a position*. The use of such general rules significantly decreases the need for adjustments when integrating new components [10].

A typical query posed to the reasoning module is: "Given a received event e, which actions are triggered by that event?" To answer this query, the integration rules are evaluated, which may have conditions such as "if the event is performed with a component of type `c`" or "if a component of type `c` is in a state `s`". Thus, the reasoning module requires information about the system's current state, such as the components that are currently active and the information objects they currently display. While the reasoner's T-Box (which defines the categories of components, events, and data objects) as well as the set of integration rules remain constant and can thus be initialized at system startup by the T-Box connector, the reasoner's A-Box (which defines the actual instance data, i.e. the set of currently active UI components and the data objects they process) is subject to constant change. Therefore, the *A-Box connector* has to work differently from the T-Box connector.

In our previous work, we have shown that pull-based approaches where the reasoner dynamically queries the integrated components' state clearly outperform push-based approaches where the integrated components notify the reasoner about state changes. The latter even turned out not to scale at all to a larger amount of components with frequent state updates [11]. Thus, we have implemented our framework with a pull-based approach, using an *A-Box connector* which collects instance data on the fly during a query by addressing the containers' components and state management services. Those services can also cache the instance data and thus further improve the system's performance.

3. ANNOTATING AND PROCESSING EVENTS

Events that are exchanged between components have to be commonly and unambiguously understood by each component. To that end, mechanisms for bridging both the syntactic (i.e. programming language) as well as the semantic (e.g. using different class or attribute names) gaps are needed. Annotation using formal ontologies can ensure such a mutual understanding [16].

As discussed in [1], RDF can be used as an interlingua bridging the syntactic gap between different components, and when using ontologies for defining the RDF elements, the semantic gap is bridged as well: events can be properly serialized and deserialized, e.g. using RDF-XML, and the meaning of each element of the serialization is defined in a formal ontology.

We use the ontology of user interfaces and interactions for defining event types. Each event notifies about an action. Our ontology contains a categorization of actions performed by the user (such as selecting, dragging and dropping objects) as well as by the system (such as highlighting, displaying or removing objects). Together with the annotation of the involved information objects and components (see below), an event can be encoded and transmitted in RDF, which allows the unambiguous decoding of an event.

4. ANNOTATING UI COMPONENTS

A container's A-Box connector has access to different type of information about the running system: UI components and the data objects they process. To be able to process those information, they need to be annotated by using the common ontologies. For each component, the *components and state management service* provides such annotation by keeping track of the component and its sub-components. It can serve the reasoning engine information about the currently active components, their respective states, and the information objects they currently process, e.g. in RDF.

The annotations can use concepts from the domain ontologies, as well as from the respective components's application ontology, e.g.

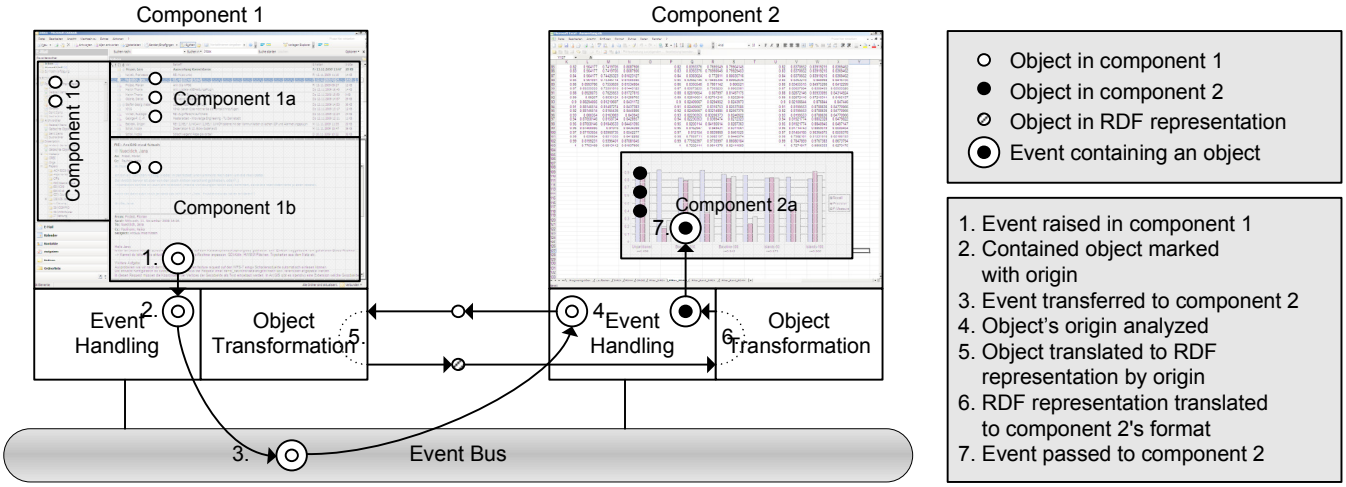

Figure 2: Schematic view of object transformation using an abstract intermediate format in an event exchange between two UI components. For reasons of simplicity, the event processing step by the central reasoner is not shown in this diagram.

for specialized components. For example, a `special button` concept can be defined in an application ontology, declaring it as a subconcept of the general `button` concept in the UIs and interactions ontology.

The resulting annotation of components is used two-fold: for inclusion in the event annotation (i.e., for defining the components that have raised an event) and for providing information about the currently active components to the central event processor through the A-box connector.

5. ANNOTATING AND PROCESSING DATA OBJECTS

Events most often involve data objects. For example, a select event does not only carry the information that the user has selected *something*, but also more information about that *something*, i.e., the selected object. In a heterogeneous system, those objects may be implemented in different programming languages and exist in different technical environments. For event exchange, those objects have to be transferred between heterogeneous components and transformed into their respective formats.

The big picture of object exchange is shown in Fig. 2: When an object is sent along with an event, the event contains a reference to the originating container and an ID to identify the object within that container. When the event is received by another component, it requests the object's abstract RDF representation. By following the reference to the originating container, that container's object transformation service is used to dynamically provide the RDF representation based on the object's ID, and the receiving container's object transformation service can translate the object back to its own representation.

The use of annotated events as a communication paradigm between integrated UI components leads to a modular system built from loosely coupled components: the only dependencies shared between the integrated components are the common ontologies, and there are no direct dependencies between components.

6. RELATED WORK

UI components can be integrated with different approaches, such as component based and plugin based systems, web portals, and mash-ups. Most of those approaches use the exposed APIs of the integrated components, and the developer has to write code invoking one component from another one. This often leads to code-tangling and to integrated systems which are not loosely coupled, not modular, and therefore hard to maintain. As discussed in [3], the means for seamless integration, as defined in this paper, are still rather limited, and formal models, which could remedy the problem, are still not used very widely. Furthermore, despite the large variety of integration approaches, only a few of those approaches explicitly address problems of cross-technology integration.

Component and plugin based approaches most often expect the plugins to follow a certain architectural style and programming language and are thus not suitable for integrating heterogeneous components. There are, however, a few examples explicitly directed at heterogeneous components, such as the *OpenInterface* workbench [6]. This platform allows components implemented with different technologies to communicate via exposed interfaces. The approach described in [5] focuses on integrating widgets from heterogeneous widgets platforms by using abstract models of those widgets and combines those models with MDA methods.

The *CRUISe* integration framework [14] uses the notion of *user interface services* for integrating different user interfaces, including Flex and HTML/JavaScript based components. Using a user interface definition language, they also focus on dynamic retrieval and binding of components, which is out of scope of our approach. The *Mixup* approach shown in [7], which is probably the closest to the one shown in this paper, uses wrappers for components implemented with different technologies, which are responsible for data conversion and communication.

Portals and mash-ups are typical approaches to UI integration on the web. The 2.0 version of the JSR portlet standard also offers a basic event processing mechanisms between portlets, including user-defined events. However, event and object exchange in most portal implementations is based on naming conventions. Thus, the portlet standard per se does not provide semantic event exchange, and does not prevent code tangling between portlets.

Mash-ups differ from portals as they use less standardized, lightweight frameworks and rather address end users and semi-professional programmers than professional software developers. Event-based mashup platforms are good candidates for implementing linked views on data by linking, e.g. integrating Google Maps with other applications to show the geographic location of selected ob-

jects. As for portals, no abstraction layer between mashlets exists that prevents code tangling when implementing cross-mashlet interaction.

7. CONCLUSION AND OUTLOOK

Composing user interfaces from existing UI components helps saving engineering time and efforts. Especially when re-using heterogeneous components together in one system, implementing cross-component interaction leads to code-tangling and close coupling, thus complicating the maintenance of the integrated user interface.

In this paper, we have introduced a framework which is capable of integrating heterogeneous UI components and follows the paradigm of loose coupling. Components can exchange events annotated using RDF and ontologies, which are processed by a central reasoner based on integration rules. In [11], we have shown that the event processing times of our framework are reasonable even for larger number of components and integration rules.

The introduction of the semantic event processor as an indirection in the event exchange process helps keeping the integrated components modular and separated from each other. By formulating the interaction rules based on ontologies, components may be exchanged for others without causing any changes on other components.

The framework introduced in this paper has been successfully used for building the SoKNOS application [12], a prototype system for emergency management comprised of more than 20 different components developed with Java and Flex.

Currently, the automatic annotation of data objects as described in this paper is limited to class models which can be mapped to an ontology in a way that each data object class and attribute maps to one concept in the corresponding ontology. We are currently extending this mechanism in a way that it also works in cases where such 1:1 mappings are not possible.

Although the run-time reasoning based on annotations, ontologies, and integration rules automatizes much of the integration and frees the developer from writing a lot of glue code, there is still a number of steps that the developer who performs the integration with our framework has to conduct. Providing an easy-to-use tool could further automatize UI integration.

8. ACKNOWLEDGEMENTS

The work presented in this paper has been partly funded by the German Federal Ministry of Education and Research under grant no. 01ISO7009 and 01IA08006.

9. REFERENCES

[1] David Booth. RDF and SOA. In *Workshop on Web of Services for Enterprise Computing*. W3C, 2007. `http://www.dbooth.org/2007/rdf-and-soa/rdf-and-soa-paper.htm`.

[2] Gaelle Calvary, T. C. Nicolas Graham, and Philip Gray, editors. *Proceedings of The 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS)*. ACM, 2009.

[3] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, and Regis Saint-Paul. Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007.

[4] Sofie Goderis, Dirk Deridder, and Ellen Van Paesschen. DEUCE : Separating Concerns in User Interfaces. In *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007), August 25-31, 2007, Cap Esterel, French Riviera, France*, page 51. IEEE Computer Society, 2007.

[5] Dimitrios Kotsalis. Managing Non-Native Widgets in Model-Based UI Engineering. In Calvary et al. [2], pages 313–316.

[6] Jean-Yves Lionel Lawson, Ahmad-Amr Al-Akkad, Jean Vanderdonckt, and Benoît Macq. An Open Source Workbench for Prototyping Multimodal Interactions Based on Off-The-Shelf Heterogeneous Components. In Calvary et al. [2], pages 254–254.

[7] Florian Danieland Maristella Matera. Mashing Up Context-Aware Web Applications: A Component-Based Development Approach. In *WISE '08: Proceedings of the 9th international conference on Web Information Systems Engineering*, volume 5175 of *LNCS*, pages 250–263, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202, New York, NY, USA, 1992. ACM.

[9] Heiko Paulheim. Ontologies for User Interface Integration. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 973–981. Springer, 2009.

[10] Heiko Paulheim. Ontology-based Modularization of User Interfaces. In Calvary et al. [2], pages 23–28.

[11] Heiko Paulheim. Efficient Semantic Event Processing: Lessons Learned in User Interface Integration. In Lora Aroyo, Grigoris Antoniou, Eero Hyvönen, Annette ten Teije, Heiner Stuckenschmidt, Liliana Cabral, and Tania Tudorache, editors, *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2010, Heraklion, Crete, Greece, May 31-June 4, 2010, Proceedings*, volume 6089 of *LNCS*, pages 60–74. Springer, 2010.

[12] Heiko Paulheim, Sebastian Döweling, Karen Tso-Sutter, Florian Probst, and Thomas Ziegert. Improving Usability of Integrated Emergency Response Systems: The SoKNOS Approach. In *Proceedings "39. Jahrestagung der Gesellschaft für Informatik e.V. (GI) - Informatik 2009"*, volume 154 of *LNI*, pages 1435–1449, 2009.

[13] Heiko Paulheim and Atila Erdogan. Seamless Integration of Heterogeneous UI Components. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2010)*, pages 303–308. ACM, 2010.

[14] Stefan Pietschmann, Martin Voigt, Andreas Rümpel, and Klaus Meißner. CRUISe: Composition of Rich User Interface Services. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proceedings of the 9th International Conference on Web Engineering (ICWE 2009)*, Edition 5648, pages 473–476, San Sebastian, Spain, June 2009. Springer.

[15] Kia Teymourian and Adrian Paschke. Towards semantic event processing. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–2, New York, NY, USA, 2009. ACM.

[16] Utz Westermann and Ramesh Jain. Toward a Common Event Model for Multimedia Applications. *IEEE MultiMedia*, 14(1):19–29, 2007.