EnviroInfo 2004 (Geneva)
Sh@ring – EnviroInfo 2004
 ISBN: 28 29 30 275-3

IRIS: A Framework Supporting Composition and Device-Specific Access of Software Services

Uwe Radetzki, Sebastian Mancke and Armin B. Cremers[1]

Abstract

In this paper we present a software platform supporting domain experts in building and maintaining compositions of services (business processes) as well as allowing flexible access to these services with different client devices. We will demonstrate our framework by means of a case study in the area of disaster management. The proposed concepts are currently developed and investigated in the research project *Interoperability and Reusability of Internet Services* (IRIS).

1. Introduction

In disaster management, time is the crucial factor. Every year wildfires or floods bring destruction, injury, and even death. The ability of incident managers to control disasters quickly, thereby limiting the damage, depends to a great extent on their ability to gather and combine information from various sources, to deploy resources, and to perform emergency evacuations efficiently (Radetzki 2002). Thus, the acquisition, processing and analysis of geo-related information is vital for disaster management (Bernard 2003). Furthermore, proper information has to be available to different units supporting them in decision-making, too. Therefore, user interfaces for different devices with different capabilities have to be made available very quickly.

In the context of Semantic Web and middleware, service-oriented architectures (SOA) are discussed as a key technology to achieve the composition of different sources. Especially the Web Service technology is a candidate for building modern software services (Cubera 2002). In the context of geo information systems (GIS) SOAs are based on standardized spatial data infrastructures (SDI). These standards are defined by organizations like the OpenGIS Consortium (OGC) or the International Organization for Standardization (ISO) and provide the basis for syntactic interoperability. But they lack solutions for semantic interoperability, nor can they support highly-sophisticated applications.

[1] Dept. of Computer Science III, University of Bonn, Roemerstr. 164, D-53117 Bonn,
email: {ur, mancke, abc}@iai.uni-bonn.de

The availability of suitable user interfaces for different devices suggests an automatic generation of these clients. Model driven approaches are discussed and researched for the generation of user interfaces (Schlungbaum/Elwert 1996). Today, this aspect gains more interest in connection with model driven architectures (OMG/MDA). The key idea is to generate executable code on the basis of declarative models. In the context of model-based user interfaces the term interaction object (IO) is defined. These objects can be divided into abstract interaction objects (AIO) and concrete interaction objects (CIO) (Vanderdonckt/Bodart 1993). However, recent approaches try to concretize a previously defined abstract representation of a user interface (Luyten/Vandervelpen/Coninx 2002). But these approaches assume a fixed correlation of AIOs and CIOs, i.e. CIOs have to correspond to AIOs and these CIOs have to be available on every device. An automatic identification, i.e. a *mapping* of available CIOs fulfilling the demands of the AIOs is not addressed, nor are there any possibilities offered that allow a dynamic upgrade of new CIOs.

2. Use Case Scenario

Consider a use case scenario of fire fighting. A group of firemen fighting against a fire source in an unknown woodland has to trust in several geographical information as well as in instructions from their headquarter leaders. This information must provide an adequate view of the landscape they are working in and should contain river and fire lines, buildings, etc. Further, the safety and well-being of all fire fighting resources on the fire line, as well as their ability to effectively fight the fire, depend on up-to-date information about current fire sources, contact with immediate supervisors, knowledge of their actual location along the fire line, and prediction of fire behavior based on fire source information combined with current weather and wind information. On the other hand, firemen may have information about a changed situation on site, e.g. altered fire lines or even new fire sources. Thus, they should be able to supply the headquarters with new information. Obviously action forces on site use different hard- and software platforms than incident managers in the headquarters do. Therefore an effective support for different client devices should be available. In the following sections we will discuss our approach based on this use case scenario.

3. IRIS Framework

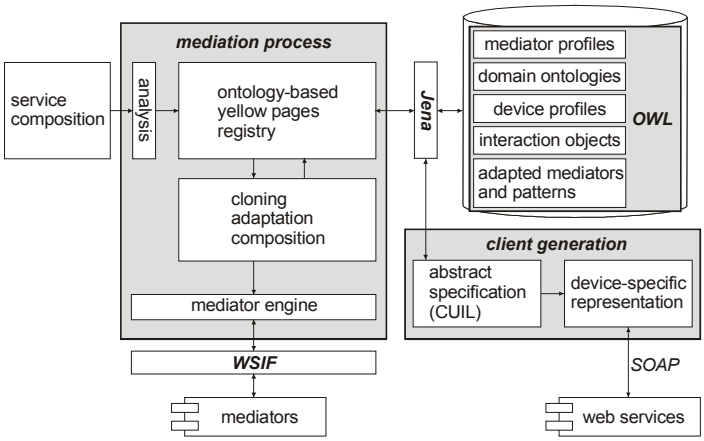
IRIS (***I****nteroperability and* ***R****eusability of* ***I****nternet* ***S****ervices*) is a web service framework addressing the creation and maintenance of service compositions, i.e. business processes, focusing on the interoperability subject as well as the access to these services with different devices. The IRIS framework can be divided into three main parts (Figure 1): The first part is responsible for the mediation process – the process

that addresses the interoperation of autonomous and heterogeneous services. The second part allows the generation of device-specific client interfaces. The last part contains the database backend. Here relevant information about the mediators, the interaction objects and the domain ontologies are stored. Both the components of the mediation process and the components of the client generation access this database backend. We further take advantage of the open frameworks Jena (OWL (W3C) and RDF(S) (W3C) processing) and the Web Services Invocation Framework (WSIF) (dynamic invocation of mediators). In the next paragraphs we will describe the concepts and modules of the mediation process and the client generation in more detail.

3.1 Sharing services – the mediation process

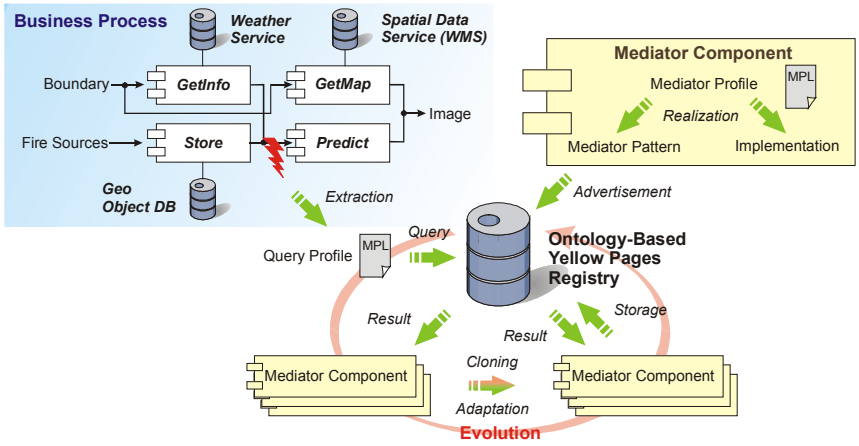
In this section we discuss the question: What is the mediation process and why is it a 'process' at all? While a domain expert defines a business process, i.e. a service composition, an implicit 'behind the scenes' *process* is started. This process – we call it *mediation process* – is responsible for overcoming the semantic and syntactic heterogeneity of the services involved and should be executed automatically as far as possible. That way domain experts can focus on "real stuff", i.e. the business process.

Figure 2 illustrates in the upper left corner a service composition for predicting the new fire line within a given boundary in the case of fire fighting. The result of executing this composition should provide the user with an image of the changed



1. Figure 1
Architecture of the IRIS Framework

situation. The composition encompasses a weather service that provides information



1. Figure 2
Mediation process

about wind direction, wind force and temperature, a WMS-like service that supports spatial information and returns a raster image of a given area, a geo service, where users can store spatial information, e.g. new objects for fire sources, and a fire prediction service that uses information of fire sources and wind forces in order to predict the new directions and dimensions of these fire sources. Obviously the interfaces and data types of these autonomous services may differ. For instance services use different coordinate systems or different scales, provide different data formats, like GML or proprietary XML dialects, and some of them might provide vector information while others support only raster images.

In IRIS we address these interoperability issues by creating a pool of registered – but distributed – *adaptable mediator components*. Mediator components represent adaptable function families that are able to *translate* between semantically and syntactically different services. The adaptability of these mediators encompasses three facets: (a) changing behavior based on specialized properties, (b) composition of mediator components, and (c) exchange of mediators by new or advanced mediator components. Because mediator components can be built by third parties, highly sophisticated transformations can be reused by adapting them for a specific aim.

Every mediator component is defined by a mediator profile which declares the component's capabilities (cf. Fig. 2, upper right corner). For that purpose we have developed an OWL-based *mediator profile language* (MPL). Knowledge of the mediator capabilities is required in order to support (a) appropriate selection of mediators, (b) adaptation within the current mediation process, and (c) activation through the *mediator engine* (cf. Fig. 1).

We have divided the capabilities of a mediator component into two parts: *functionalities* and *properties*. Functionalities define the mediation features of a mediator component, i.e. they represent its executable part, while properties modify the behavior of the functionalities. For instance, assume a 'coordinate transformation' functionality, where a 'coordinate system' property allows the specification of the result's target coordinate system.

Regarding the realization of a mediator component, it can be directly implemented in a programming language like Java (*atomic mediator component*), or it can be created by assembling together other mediator components (*mediator pattern*). The mediator patterns are declared in MPL through the definition of data flow graphs and control flow graphs, where the nodes represent functionalities of mediator components (Radetzki/Bode/Cremers 2004).

With a given initial number of advertised mediator components the mediation process is designed as follows: In a first step the service composition is *analyzed*, i.e. the WSDL descriptions (W3C) of data-producing and data-consuming services are extracted in order to generate a description of the required mediator component (*query profile*). This query profile is defined in MPL and can be further modified or enhanced by user intervention. In a next step a specialized retrieval unit – the *ontology-based yellow pages registry* – matches the query profile with the mediator profiles of advertised mediator components and offers suitable mediator components. Potential mediator patterns are considered, too. Detailed information of the registry unit can be found in (Radetzki/Cremers 2004).

Now the identified mediator components and mediator patterns are presented to the user and the adaptation phase begins. Here, the qualities of the component-based approach provide the desired flexibility and reusability of mediator components in different contexts (Szyperski/Gruntz/Murer 2002). First the user *selects* suitable mediator components which are cloned for further adaptation. Then the user can *modify* properties or/and mediator patterns. Changing a pattern means to modify the data flow or control flow but also to replace a mediator component with another one. Sometimes it can be important to create new functionalities or mediator patterns. For example, if a WFS-service provides GML-based data, but an available mediator component processes only the feature information, then a projection is required before the mediator can be executed. This projection can for instance be defined using XQuery (W3C). The definition of this kind of mediator component can be done interactively or (under some assumptions) automatically. For instance, (Bowers/ Ludäscher 2004) represent an approach which relies on a formalized ontology and registration mappings, in order to automatically generate required data transformations. At the moment, adaptation of properties and creation of projection mediators as described above are done by the user with system support, but in the future more intelligent mechanisms will be integrated in IRIS.

After the adaptation phase the modified mediator components are advertised to the registry, so that they are directly available for future (similar) application contexts. That way the knowledge of the registry grows with every mediation process and *evolves* over time and more advanced mediator components become available.

3.2 Sharing user interfaces – device-specific client generation

In order to cope with a heterogeneous set of user devices, like outlined in the firefighter scenario (clients differing in a wide range of hard- and software capabilities), one requires independent user interface (UI) realizations for each target platform. But separately developing a UI for every device which meets all device-specific restrictions, is very lengthy and time-consuming. Further, new client devices cannot be included short-term, which might be a problem in scenarios like disaster management. Maintaining different realizations of the 'same' UI for different devices is also error-prone and cumbersome. Obviously the development effort raises linear with each adventitious client device and thus quadratic with adventitious client devices and software features. We address this problem by a *two-phase adaptation* technique, which enables us to automatically transform a single abstract UI definition into executable UIs on multiple client platforms (Mancke 2003). Figure 3 illustrates this kind of client generation.

The generation of device-specific UIs starts with an abstract description of the requirements and capabilities of the UI and is specified by an XML-based *Component User Interface Language* (CUIL) (Mancke 2003). Such a description includes visual and pure logical (non-visual) components. Visual components encompass objects for user interaction, so-called *interaction objects* (IOs), as well as *structure elements*, which are used to group and arrange the IOs. The logical components cover control components, data definitions and service declarations. They can be used for the definition of event handlers, which are connected to visual components, and interact with web services, process data or modify visual components.

The visual components are defined in an absolutely abstract manner. For this purpose we refined the concept of *abstract interaction objects* (AIOs), originally introduced by (Vanderdonckt/Bodart 1993), where the semantic of these abstract components is defined by an ontology. In CUIL an AIO is a semantic representative for capabilities, so-called *IO-functionalities*, that the user interface provides for user interaction. An AIO can be regarded as a requirement list of the IO-functionalities, which may, should or have to be available in a later executable realization of the UI. For example, an AIO could require that "there *has to* be an interaction object that allows the selection of coordinates from a list of geographical points". Additionally the IO-functionalities of AIOs determine the data requirements and event capabilities of the interaction object, so that a *binding* from AIOs to the logical components

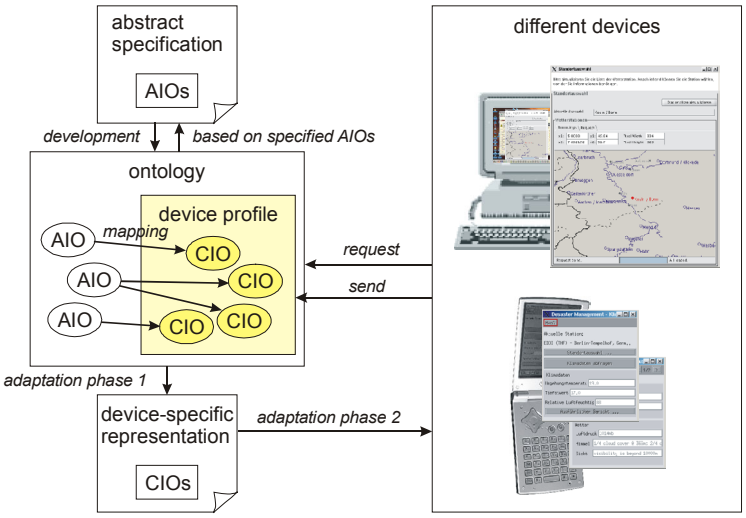


Figure 3
Device-specific client generation

can be performed. The vocabulary of AIOs and their IO-functionalities is arbitrarily extensible and is determined by a formal, RDF-based ontology.

The structure elements are used to combine and arrange multiple AIOs. They are similar to the AIOs themselves, but may have nested elements. To simplify usage, the set of available structure elements is predefined and fixed (i.e. application, group, menu, dialog).

The result of this description process is a specification of the UI as a composition of abstract components representing different aspects, without any realization-specific or graphical flavor. These descriptions are stored on the server side of IRIS (cf. Fig. 1).

Upon a client request the server starts the *first adaptation phase*, which transforms the abstract description to a concrete description for execution on the client. The concrete description is derived from the abstract one, by replacing the AIOs through *concrete interaction objects* (CIOs), which are suitable for the target device and meet all requirements of the AIOs. A CIO represents the real interaction object (e.g. a graphical button or a text input) existing in a realization, which is also described by its IO-functionalities. The replacement can be seen as a *mapping* from AIOs to CIOs (a CIO fulfilling all requirements of the AIO is said to be *valid*). The calculation of this mapping is based on the formal description of the AIOs and the respective description of the CIOs as well as common client properties (i.e. display size, device category, etc.). For this purpose the client transmits a description of its capabilities to the server, using an extended type of Composite Capabil-

ity/Preference Profiles (CC/PP) (W3C). Including the IO-functionality assures a resulting description that offers all the needed interaction capabilities. Additionally taking the common client properties into account allows an optimization under several aspects (e.g. display size) by *quality* functions. The AIO replacement also takes CIOs into account that are not available on the client side yet, but loadable from server side.

The resulting device-specific representation is sent to the client (cf. Fig. 3). Here, the *second adaptation phase* begins: A runtime engine renders the description in a device-specific way, i.e. calculates a suitable layout and displays it with the local GUI library. The algorithms for layout calculation vary for each device class, i.e. realize special layout strategies, in order to satisfy the different device requirements. They are designed to preserve the existing structure as good as possible, which is given by the structure elements in the description. Their major task is calculating the actual sizes of all CIOs, arrange them inside the container elements using boxes and grids, as well as splitting large container elements to distribute them across multiple screens. The structure elements of the UI descriptions are represented by container elements which are available and suitable for the client, e.g. application windows, bordered panels or menu bars.

To sum up, the adaptation process is separated into two decoupled phases. The first phase is the concretion of the AIOs by a concrete, structured UI, lacking all display information. The second phase transforms this structure into layout to get a displayable user interface. This separation has two main advantages: First, adaptation decisions, which are critical for the functionality of the user interface can be controlled in one central place (server side). Second, the layout specific processing can be done at runtime on the client, when all necessary parameters are known (font settings, exact size of widgets with data, etc.).

4. Conclusion

In this paper we address two major topics in the field of SOA: on the one hand we focus on interoperability issues, which occur in creating service compositions (i.e. business processes), on the other hand we propose a solution for automatic client generation for accessing web services through different client devices.

The interoperability issues of service composition is addressed by a mediation process. Here, in a first step the interface descriptions of data-producing and data-consuming services are explored and a query profile is automatically generated. This profile is in a next step compared with mediator profiles of previously advertised mediator components and suitable mediators are retrieved fulfilling the requirements as far as possible. In a third step the identified mediator components are cloned and adapted with respect to the current mediation context. Finally the adapted mediator components are stored for future retrieval sessions. In this way the registry, sup-

porting advertising and retrieval of mediator components, grows and evolves over time, thus future service compositions can benefit from enhanced and adapted mediator components.

The generation of device-specific user interfaces is done by a two-phase adaptation technique. A software developer specifies a user interface abstractly as a composition of AIOs, whose semantic is defined by an ontology approach. Afterwards concrete representatives (CIOs) corresponding to a specific device are determined (adaptation phase 1). This concretion is based on quality functions and validity functions we have developed that consider both the AIOs and the device capabilities available through device profiles. Finally, this more concretized description is executed on the specific client device, where a suitable layout is calculated (adaptation phase 2).

Acknowledgement

The authors want to thank Thomas Erdenberger and Michael Schaefers for valuable discussions and fruitful comments.

Bibliography

Bernard, L. (et al) (2003): Ontologies for Intelligent Search and Semantic Translation in Spatial Data Infrastructures, Photogrammetrie - Fernerkundung - Geoinformation (PFG), No. 6, pp. 451-462

Bowers, S., Ludäscher, B. (2004): An ontology-driven framework for data transformation in scientific workflows. International Workshop on Data Integration in the Life Sciences (DILS'04)

Cubera, F. (et al) (2002): Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI, IEEE Internet Computing, 6(2), pp. 86-93

Luyten, K., Vandervelpen, C., Coninx, K. (2002): Migratable User Interface Descriptions in Component-Based Development, Workshop on Design, Specification and Verification of Interactive Systems (DSVIS-2002)

Mancke, S. (2003): Specification and Adaptation of Service-Oriented User Interfaces, Master Thesis, University of Bonn, (in German)

OMG Model Driven Architecture (MDA), Object Management Group

Radetzki, U., Cremers, A. B. (2004): IRIS: A Framework for Mediator-Based Composition of Service-Oriented Software, Proceedings of the IEEE International Conference on Web Services (ICWS 2004), San Diego, USA, pp. 752-755

Radetzki, U., Bode, T., Cremers, A. B. (2004): Mediatorbasierte ad hoc Integration autonomer Web Services. Informatik 2004, 34. Jahrestagung der Gesellschaft für Informatik (GI), Workshop: Dynamische Informationsfusion, Ulm, Germany

Radetzki, U. (et al) (2002): First Steps in the Development of a Web Service Framework for Heterogeneous Environmental Information Systems, Proceedings of the 16th International Conference "Informatics for Environmental Protection" (EnviroInfo 2002), Vienna, Austria, pp. (1)384-391

Schlungbaum, E., Elwert, T. (1996): Automatic User Interface Generation form Declarative Models, Workshop of Computer-Aided Design of User Interfaces (CADUI-1996)

Szyperski, C., Gruntz, D., Murer, S. (2002): Component Software: Beyond Object-Oriented Programming. Component Software Series. ACM Press, 2nd edition

Vanderdonckt, J., Bodart, F. (1993): Encapsulating Knowledge For Intelligent Automatic Interaction Objects Selection, Proceedings of the ACM Conference On Human Factors in Computer Systems (INTERCHI-1993), ACM Press

W3C Specifications (2001-2004): Web Services Description Language (WSDL), Composite Capability/Preference Profiles (CC/PP), Resource Description Framework (RDF), Web Ontology Language (OWL), XML Query (XQuery)