Application Integration on the User Interface Level: an Ontology-Based Approach

Heiko Paulheim and Florian Probst

SAP Research Center Darmstadt

{heiko.paulheim,f.probst}@sap.com

Abstract. Integration of software applications can be achieved on different levels: the data level, the business logic level, and the user interface level. Integration on the user interface level means assembling the user interfaces of existing applications in a framework allowing seamless, unified interaction with those applications. While integration on the user interface level is desirable both from an software engineering as well as from a usability point of view, most current approaches require detailed knowledge of the integrated applications and make the implementation of a common interaction that involves different applications a difficult issue.

In this paper, we propose a framework using ontologies for application integration on the user interface level by encapsulating the applications in plugins. Our approach is to use different ontologies for characterizing applications and the interactions possible with them, and for semantically annotating information objects exchanged between applications. Thus, the domain-independent and the domain-specific parts are untangled, which makes the framework applicable to different domains. An instance-based reasoner is used to process the ontologies and to compute the possible interactions, thus enabling integration at run-time.

In an example from the domain of emergency management, we show how our approach helps implementing cross-application interactions more easily, thus significantly lowering the barriers for interoperability.

Application Integration on the User Interface Level: an Ontology-Based Approach

Heiko Paulheim, Florian Probst

SAP Research Center Darmstadt, Bleichstrasse 8, 64283 Darmstadt, Germany

Abstract

Integration of software applications can be achieved on different levels: the data level, the business logic level, and the user interface level. Integration on the user interface level means assembling the user interfaces of existing applications in a framework allowing seamless, unified interaction with those applications. While integration on the user interface level is desirable both from an software engineering as well as from a usability point of view, most current approaches require detailed knowledge of the integrated applications and make the implementation of a common interaction that involves different applications a difficult issue.

In this paper, we propose a framework using ontologies for application integration on the user interface level by encapsulating the applications in plugins. Our approach is to use different ontologies for characterizing applications and the interactions possible with them, and for semantically annotating information objects exchanged between applications. Thus, the domain-independent and the domain-specific parts are untangled, which makes the framework applicable to different domains. An instance-based reasoner is used to process the ontologies and to compute the possible interactions, thus enabling integration at run-time.

In an example from the domain of emergency management, we show how our approach helps implementing cross-application interactions more easily, thus significantly lowering the barriers for interoperability.

Key words: Ontologies-based System Integration, Plugin-based Software, Ontology-Driven User Interfaces

1. Introduction

In the past decades, software engineering has been changing from building one-piece monolithic programs to assembling systems from modular and distributed components. The idea of developing software in a modular fashion is nearly as old as software engineering itself and goes back to the late sixties [1]. Furthermore, as the number of pre-existing software components is continuously increasing, integration of those components has become an important part of software engineering [2]. Today, developers and integrators should be enabled to connect software modules without having to know about other components' internal functionality except for well-defined interfaces. Reality, however, most often shows a different picture: integrating software applications requires large coordination efforts, and component developers and integrators need a lot of knowledge about how other components actually work internally.

Integrating applications requires a common understanding of those applications as well as the information they process. Such a common understanding can be made explicit using ontologies, which are then used to formally describe the behavior of the systems to be integrated, and the semantics of their data. The process of integrating systems and their data by employing explicit representations of their semantics is often referred to as *semantic integration* or *ontologies-based integration*.

Software systems most often consist of three layers: the data source layer, the business logic layer, and the user interface layer [3]. Therefore, the integration of software systems can be performed on three levels, as shown in Fig. 1.

System integration can be performed on different levels [4]. Figure 1, following the work described in [5], shows those levels:

Email addresses: `heiko.paulheim@sap.com` (Heiko Paulheim), `f.probst@sap.com` (Florian Probst)

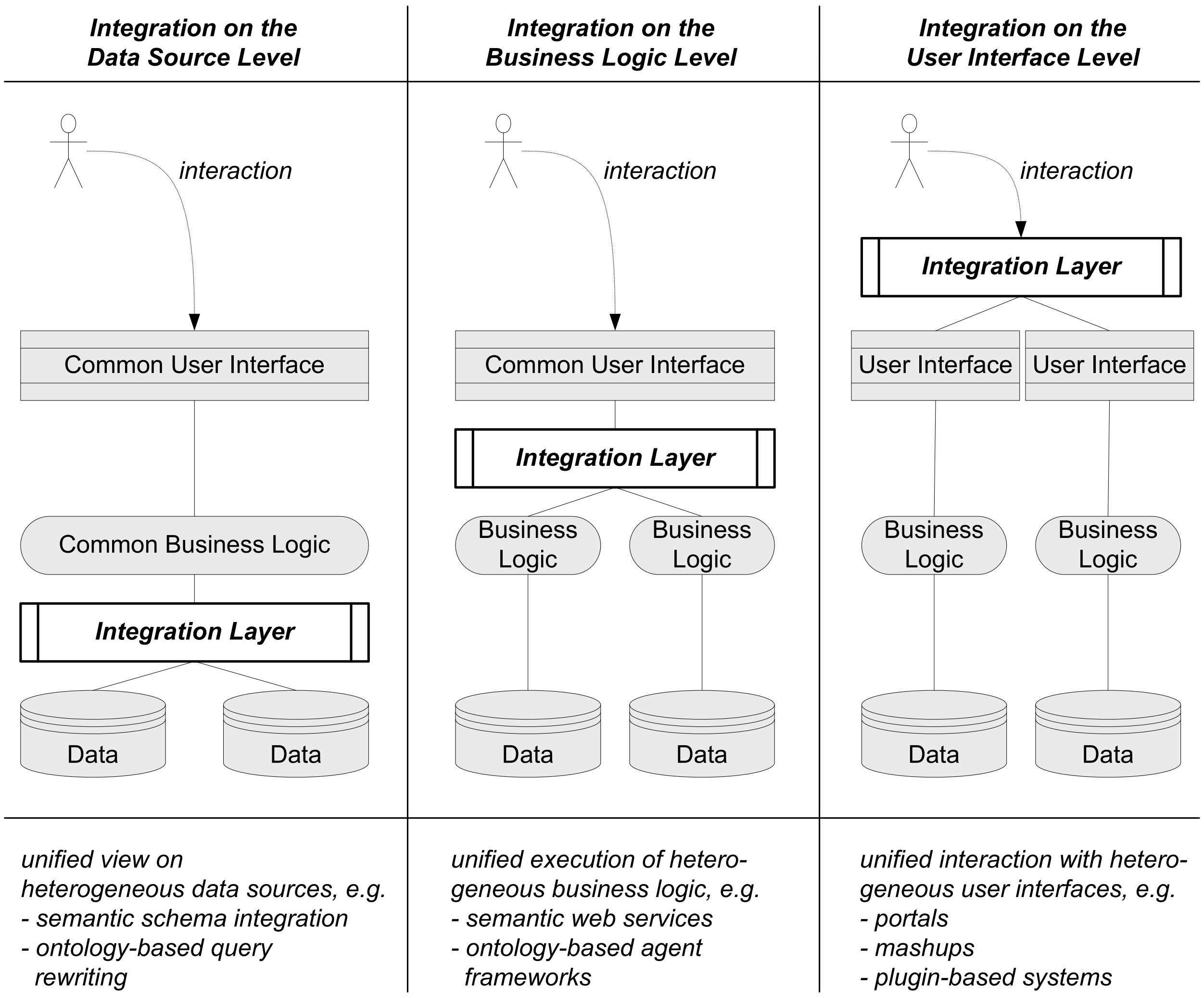
Integration on the Data Source Level
Integration on the Business Logic Level
Integration on the User Interface Level
interaction
interaction
interaction
Common User Interface
Common Business Logic
Integration Layer
Data
Data
Common User Interface
Integration Layer
Business Logic
Business Logic
Data
Data
Integration Layer
User Interface
User Interface
Business Logic
Business Logic
Data
Data
unified view on heterogeneous data sources, e.g.
- semantic schema integration
- ontology-based query rewriting
unified execution of heterogeneous business logic, e.g.
- semantic web services
- ontology-based agent frameworks
unified interaction with heterogeneous user interfaces, e.g.
- portals
- mashups
- plugin-based systems

Figure 1: Different levels of systems integration, following [5]

- *Integration on the data source level* provides a unified view on heterogeneous data sources. On this level, ontologies have been used to describe those data sources and schemas. Such an ontology-based description allows approaches such as providing a unified querying interface based on an ontology, where an integration layer rewrites the queries to and the answers from the different underlying data sources [6].

- *Integration on the business logic level* unifies different implementations of business logic, each using its own data sources, under a common user interface. On this level, ontologies have been used to describe business logic functionality. The most prominent approaches on this level are semantic web services [7] and ontology-based agents [8]. Ontologies-based integration on this level is also referred to as *semantic enterprise application integration (EAI)* [2].

- *Integration on the user interface level* unifies different user interfaces in one common system, e.g. a portal or a plugin-based user interface. Although several approaches exist for integration on the user interface level [5], little work has been undertaken so far for *ontology-based* integration on that level.

In the first two cases, a common user interface for the integrated systems is developed, which means that existing user interfaces are discarded. For example, in web-service based approaches, existing business logic is encapsulated in web services, which is then invoked from a newly developed common user interface. However, developing new user interfaces is not desirable from a software engineering point of view, since developing a system's user interface consumes about 50% of the total system development time [9]. Furthermore, users already familiar with the integrated systems' interfaces will experience a steeper learning curve than with a new user interface developed on top of integrated business logic or data sources.

To facilitate ontology-based application integration on the user interface level, we follow a plugin-based approach, where each application to be integrated is encapsulated in a *plugin*. We introduce a framework to formally describe those plugins. Application developers formally describe their applications in application ontologies, based on domain-level ontologies as common ground. Using those ontologies, integration rules can be formulated that define how applications are supposed to interact with each other. By working with those abstractions, it is possible to develop applications independently from each other and put them together in a plug-and-play fashion. Furthermore, by inroducing a reasoner as an indirection between plugins, dependencies are reduced, and the resulting systems remains modular and maintainable.

In this paper, we present a first cut on an ontology of the domain of user interfaces and interactions, which defines the basic categories needed for formally describing user interfaces and their behavior. This ontology is used to characterize applications to be integrated on the user interface layer. It can be combined with *real world domain ontologies* defining the vocabulary of the real world domain the system is built for, such as banking, travel, etc. By separating those two types of ontologies, we draw a clear line between ontologically *characterizing the functional elements of a software component* and *defining the semantics of the information items communicated between those components*. As the real world domain ontologies are not an integral part of the framework, applications for different domains can be developed by using different real world domain ontologies without having to modify the framework as such.

The remainder of this paper is structured as follows. Section 2 gives a brief overview of the relevant concepts from the Semantic Web, ontologies, plugin-based software development, and user interface level application integration. Section 3 describes the implementation of our framework, and section 4 contains a walk-through example which explains how applications can interact in our framework. We conclude with a review of related work and an outlook on future research.

2. Background

Among the many definitions of information system ontologies, we follow the one given in [10]: "An ontology is an explicit, partial account of a conceptualization". Putting emphasis on the fact that an ontology can only partially account for a conceptualization, this definition reflects its foundations in model theory: a model can always only partially reflect or account for the modelled phenomenon. How ontologies are tied to information systems becomes even clearer in the following definitions: "[An ontology is] a set of logical axioms designed to account for the intended meaning of a vocabulary" and "an ontology is a logical theory accounting for the intended meaning of a formal vocabulary" [11]. From a more pragmatic perspective, ontologies can be distinguished depending on the expressiveness and degree of formalization resulting in light-weight and heavy-weight ontologies [12].

There are different languages for expressing ontologies. Besides the most standardized RDF Schema and OWL [13, 14], F-Logic [15] has been gaining popularity in the semantic web community [16]. For expressing more complex statements, rule extensions can be added to many of those languages, while F-Logic has native means to express rules [17].

Ontologies can be used throughout the whole cycle of software development [18, 19], starting from ontology-assisted requirements engineering [20] up to ontology-supported software maintenance [21]. In [22], a categorization for using ontologies in software engineering is proposed. The authors distinguish the modelling of domain vs. infrastructure knowledge, and the employment at development time vs. run time, thus leading to a fourfold categorization.

For consistently using ontologies in a software system, programming models are needed. According to [23], direct, indirect, and hybrid models can be distinguished. In direct models, the domain concepts are directly implemented in the programming language, e.g. as Java classes. Indirect models, such as JENA [24] or OWL API [25], provide classes for the meta model, e.g. ontology classes and properties. While direct models are considered less flexible, e.g. with regards to evolving the ontology, indirect models are more complicated to handle and lead to less intuitive programming code. Hybrid models, as proposed in [23], use a direct model for top-level domain concepts and an indirect model for bottom-level ones.

In the framework discussed in this paper, we use a plugin-based approach for user application integration on the interface level. Software plugins were traditionally conceived for developing extensible software. In classical plugin-based software, a hosting application provides a fixed number of extension points with defined interfaces where plugins implementing those interfaces can add a certain functionality to the hosting application. A plugin cannot run on its own, it always depends on a hosting application. However, unlike a component, a plugin's presence is not necessarily required for its potential hosting application to run [26]. While in traditional plugin-based systems, dependencies between plugins are possible, our framework aims at a degree of modularization where all plugins are independent of each other [27].

In the past years, *pure* plugin-based software systems have also been developed, where the hosting application consists only of an engine for running plugins not providing any specific functionality to the end user, and plugins that can define their own extension points for other plugins [28]. The most famous example for a pure plugin-based system is probably the programming environment *Eclipse* [29].

Besides plugin-based systems, there are other proposed architectures for application integration on the user interface level, such as portals [30] and mashups [31]. Most of those approaches have significant shortcomings concerning the implementation of cross-application interaction, which in most cases is rather difficult, requires deep understanding of the involved components, and leads to code tangling and non-modular, hard-to-maintain systems [5].

For improving application integration on the user interface level, it has been claimed in [32] that models of user interfaces are required that are *formal*, *modular*, *human readable*, and *simple*. While simplicity lies in the eye of the beholder, ontologies fulfil at least the first three of those criteria. Therefore, we introduce a framework using ontologies for formally describing user interfaces and improving application integration on the user interface level.

3. Framework

Our framework for integrating applications by encapsulating their user interfaces in ontologically characterized plugins is implemented in Java, using Swing for the GUI part and OntoBroker [33] for semantic processing, with F-Logic as a language for building the ontologies. As pointed out above, ontologies as well as rules are used in our framework. F-Logic has been chosen since it provides the means for expressing both in one single language, and its description logic part has more expressive power than OWL DL [17].

3.1. Ontological Characterizations and Semantic Annotations

Two separate domain-level ontologies are used for capturing the ontological nature of the integrated application's elements as well as for semantically annotating the information objects exchanged between those applications:

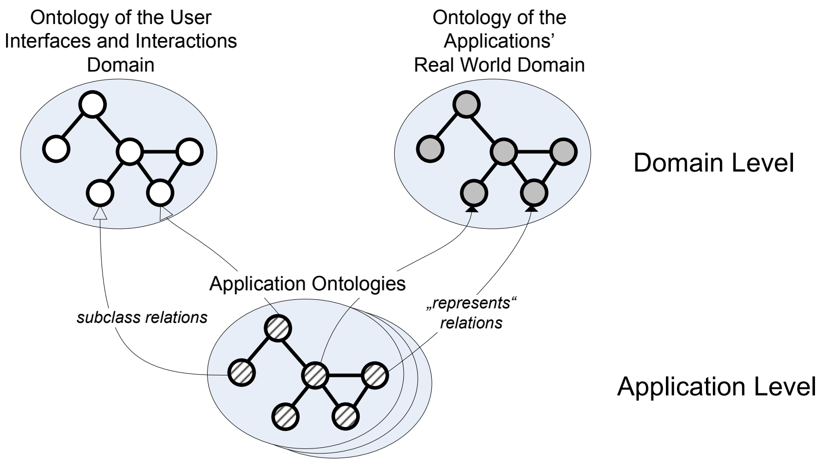
Ontology of the User Interfaces and Interactions Domain
Ontology of the Applications' Real World Domain
Domain Level
Application Ontologies
subclass relations
„represents" relations
Application Level

Figure 2: On the domain level, an ontology of the user interfaces and interactions domain and a real world domain ontology are used to define the basic categories for describing applications' user interfaces and the interactions that are possible with them. On the application level, each application is described in an application ontology: the classes defined in the applications ontologies are *subclasses* of those from the application ontology, and their instances may *represent* objects defined in the domain ontology [34].

- The *ontology of the user interfaces and interactions domain* contains basic categories such as InteractiveComponent, Interaction, UserAction, and SystemAction. In terms of [22], the application ontology is used for formalizing *infrastructure knowledge* and implementing an ontology-*enabled* architecture (OEA). Each component in a user interface belongs to a certain category in the application ontology; in this sense, elements of the applications are *categorized* with the help of the application ontology.
- The *real world domain ontology*, unlike the ontology of the user interfaces and interactions domain, is *not* an integral part of our framework and can be exchanged when developing an application for a different domain. It is also possible to use more than one domain ontology, e.g. when developing applications touching concerns from different domains, or for using modularized domain ontologies. The information objects processed by the plugins are semantically annotated by using concepts from the domain ontology. In terms of [22], the domain ontology is used for formalizing *domain knowledge* and implementing an ontology-*based* architecture (OBA).

For integrating an application in our framework, it is encapsulated in a plugin, which is responsible for displaying the application and managing event exchange with other plugins. For each integrated application, the developer creates an application ontology which defines the categories of components and interactions that are provided by the application. To this end, *subclasses* of Plugin, Interaction etc. are defined. For such descriptions, concepts from the domain ontology are used to define which objects processed by a plugin *represent* which real world objects. Fig. 2 shows how the different ontologies are related to each other.

Based on those application ontologies, integration rules for coordinating cross-application interactions can be defined. The next section describes the ontology of the user interfaces and interactions domain in more detail. An example domain and application ontology as well as their connections and corresponding integration rules are shown in section 4.

3.2. An Ontology of the User Interfaces and Interactions Domain

For explaining the necessity of the clear distinction of the ontologies on the domain level – the ontology of the user interfaces and interactions domain and the ontology of the real world domain –, we introduce a simple analogy. Consider a solar system model made of wooden balls and metal poles. Such a model can be seen as an information system displaying the movement of planets in our solar system. The model consists of wooden balls standing for the sun and the planets but also of poles holding the balls and a motor for making the balls go round. Some of the parts *represent* objects in the real word – such as the balls representing the sun and the planets – while others, such as the poles, do not. This is equally true for user interfaces: there are some objects in the user interface that *represent* objects in the real world, such as a table row entry representing a customer, and others that are not, such as a button.

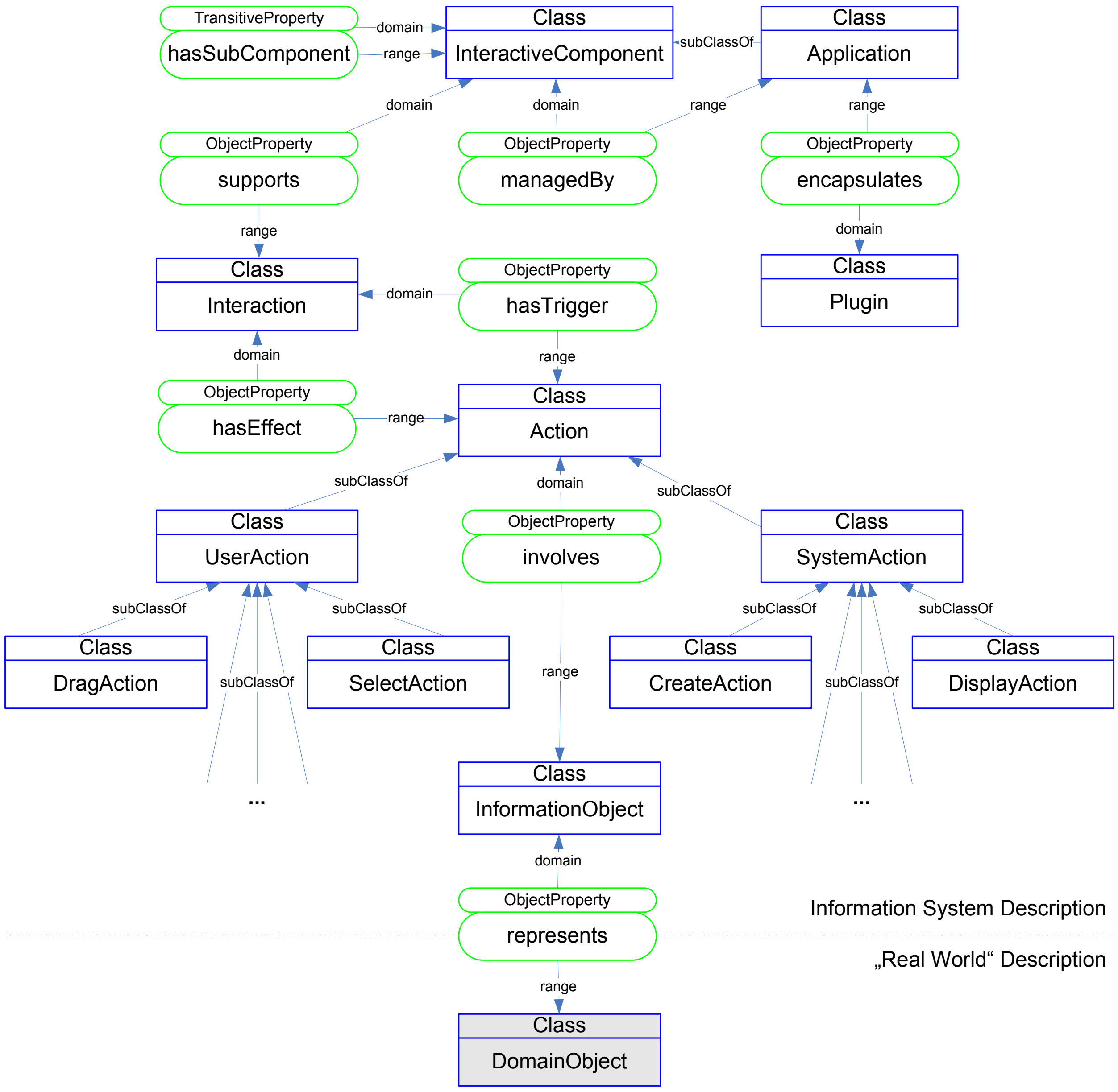

Figure 3: An excerpt of the ontology of the user interfaces and interactions domain, shown in OWL notation [14]. The central categories are INTERACTIVECOMPONENT, INTERACTION, USERACTION, and SYSTEMACTION. The REPRESENTS relation links the ontology of the user interfaces and interactions domain to the real world domain ontology.

To account for that separation of concerns, we decided to model them as two distinct ontologies: one ontology for describing the information system as such and its "technical" parts (in our analogy: the components for assembling the solar system model, i.e. the motor, the poles, and so on) and one ontology for the information system's real world domain (the solar system itself, i.e. the sun, the planets, their revolution periods and so forth). This separation facilitates the task of integrating new parts of an information system, e.g. adding new software applications to the integrated system.

Since the application ontology does not contain any domain-specific concepts and vice versa, applications for different domains can thus be built with our framework by using different real world domain ontologies, and there is no need for modifying the ontology of user interfaces and interactions to this end. In the solar system analogy, we could rearrange the parts of the model and use them as a model for an atom. In that case, we would still deal with the same wooden balls, now representing electrons instead of planets.

Besides the different concerns addressed by those two ontologies, there is also a subtle semantic difference. When the framework runs an application encapsulated in a plugin, this application *is* an instance of the category Application defined in the ontology of the user interfaces and interactions domain. In contrast, instances from the real world domain ontology's categories do not exist in the information system – all information objects processed by the plugins merely *represent* the instances from the domain ontology's categories. Therefore, the information objects – by pointing to objects in the world – carry *semantics* or *meaning*, while the instances of the application ontology, such as plugins, do not have a meaning.

The distinction of information objects and the objects in the world they represent has practical relevance when describing a plugin's functionality. To explain this relevance, we consider a real world domain ontology from catastrophe management containing the classes TacticalUnit (such as a fire brigade unit) and Order. There is an obvious difference between *creating* an *information object* for an (existing) object in the world, such as adding a new TacticalUnit to a database (which does not imply the creation of a physical fire brigade car in the world), and creating a *domain object*, such as the system issuing an Order to a fire brigade unit, which, besides creating an information object representing that order, also sends a message to the addressed unit, thus creating an instance of an Order in the world. When stating that an application *creates* objects, the strict separation of information objects and domain objects is therefore needed; the same holds for *modifying* and *deleting* objects.

Fig. 3 depicts some of the most important concepts from the ontology of the user interfaces and interactions domain. The core category is Interaction. Each interaction has a trigger and an effect, each being an action falling into one of the two sub categories User Action and System Action, which are further specified in two hierarchies of user and system actions. Each interactive component can support several interactions.

Each user action and system action *involves* information objects, which *represent* objects from the real world domain the application is designed for and hence are defined in the domain ontology. Thus, the *represents* relation links the application ontologies to the real world domain ontology and forms the bridge between the "information system world" and the "real world". It is further noteable that the two ontologies on the domain level – the ontology of the user interfaces and interactions domain and the ontology of the applications' real world domain – have no direct links to each other. Thus, they are completely independent from each other; this means that *any* real world domain ontology can be used with our framework without further modifications, and that the real world domain ontology can be exchanged for building applications for different real world domain based on our framework.

3.3. System Architecture

Our application framework, shown in Fig. 4, provides the possibility to define and execute applications which are encapsulated in plugins and augmented with an ontological characterization. Following the classical three-layer model [3], each of the integrated applications consists of three parts:

1. A GUI, i.e., each plugin is capable of displaying itself and interacting with a user.
2. Some business logic, i.e. the functionality of the application. The logic part can range from very narrow (e.g. for applications merely displaying information to the user) to arbitrarily complex logics, including calls to external components, such as web services.
3. An object model in which the data processed by the application is defined. Although it would be possible (and even simpler) to use a common object model throughout all applications, it is a more versatile approach to

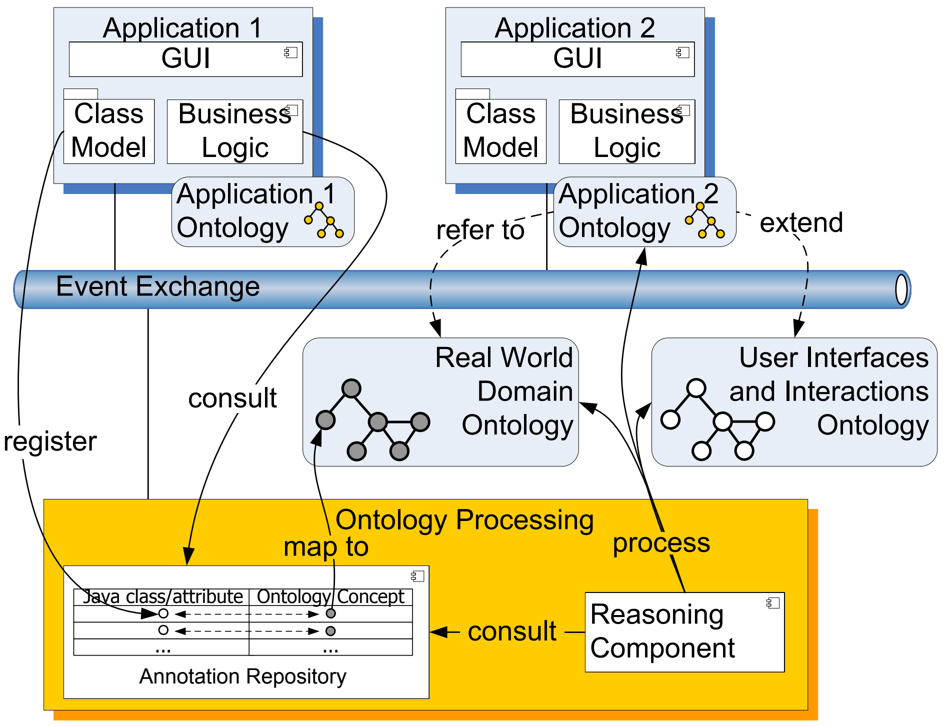

Figure 4: System architecture. Every application comes with an application ontology that extends the user interfaces and interactions domain ontology and refers some to the categories in the real world domain ontology. Applications (encapsulated in plugins) as well as a reasoning component communicate via events.

allow each developers to create their own object models fitting the respective implementation needs of their applications.

For integration, each application is described in an application ontology, which ontologically characterizes the application as well as its components and the interactions they support. The application ontology extends the ontology of the user interfaces and interactions domain by defining subcategories e.g. of Application, InteractiveComponent, and InformationObject. Since information objects represent domain objects, each information object has a represents relation to an object from the domain ontology.

Plugins can exchange events via an event exchange mechanism which allows broadcasting as well as point to point communication. These events can also consumed and produced by a reasoner component, which is responsible for processing the domain and application ontologies.

Since each application may use its own object model, mappings between that object model and the real world domain ontology are needed to provide semantic annotations of those objects at run time. In our framework, the *annotation repository* stores thoses mappings, which are registered by each plugin. Both, classes and properties (i.e. attributes accessible via getter and setter methods), can be mapped to concepts defined in the ontology. In the terminology of [23], this approach combines a direct model with an indirect one: the plugin developer can utilize a direct model's programming convenience, and the system can make use of reasoning. Furthermore, each plugin developer may use exactly the subset of categories and relations from the domain ontology required for the plugin's purpose. A mapping from a class model to the real world domain ontology consists of tuples of the form $< className, URI >$ for annotating classes, and triples of the form $< className, attributeName, URI >$ for annotating attributes, where in the first case, the URI should point to a class in the real world domain ontology, in the latter case, it should point to a property.

The reasoner introduces an indirection between applications, so that they do not have to react to events created by other applications directly. Instead, plugins broadcast their events; the reasoner reads those events and processes the integration rules to find out which applications have defined interactions triggered by that sort of events, and notifies

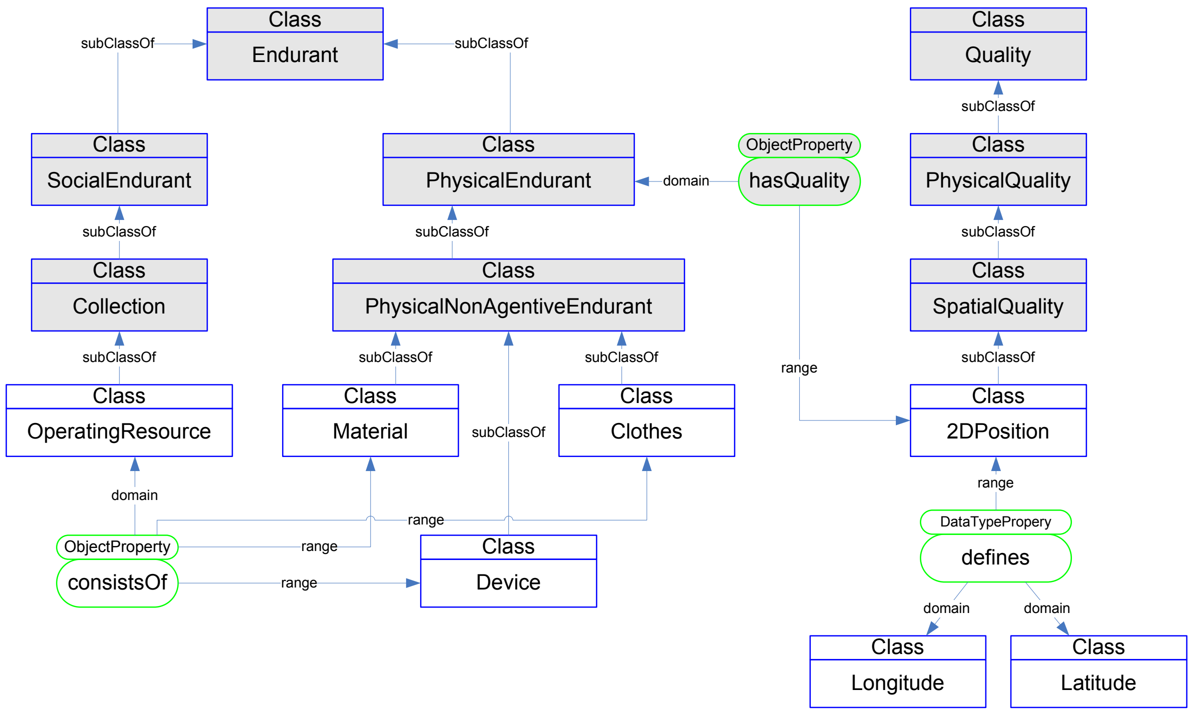

Figure 5: Excerpt of the emergency management domain ontology used in the example scenario. The ontology is grounded in DOLCE; classes and relations from DOLCE are marked in gray.

the respective plugins to perform the appropriate actions. Due to this indirection, no direct dependencies between the applications exist, and the overall system becomes more modular and more easily maintainable [27]. The next section explains the event processing mechanism in more detail.

4. Example

4.1. Scenario

In the following scenario, we introduce a setting from emergency management in which our integration framework is used. As sketched above, using our framework to build an integrated application requires a suitable real world domain ontology. Therefore, we use an ontology covering the relevant parts of the emergency management domain [35]. Fig. 5 shows an excerpt of that domain ontology, which is grounded in the foundational ontology DOLCE [36]. The concepts taken from DOLCE are marked in gray.

The scenario deals with the SoKNOS emergency management system [37]. The system consists of several applications built for dealing with resources, orders, messages, and so on. In the following examples, we focus on two applications (see Fig. 6): one is a geographic information (GI) application capable of displaying maps and layers with objects, the other shows an inventory of resources, such as material or devices, in which the user can find and select resources and edit their basic properties.

Fig. 6 indicates a *linking* interaction [38] between those plugins: if the user selects an object in the inventory list, the map component highlights the corresponding symbol on the map. We show the process of integrating the GI application in our framework.

4.2. Preparation

Three steps are necessary for integrating an application in our framework:

1. describe the interaction of the applications in the respective applications ontologies,

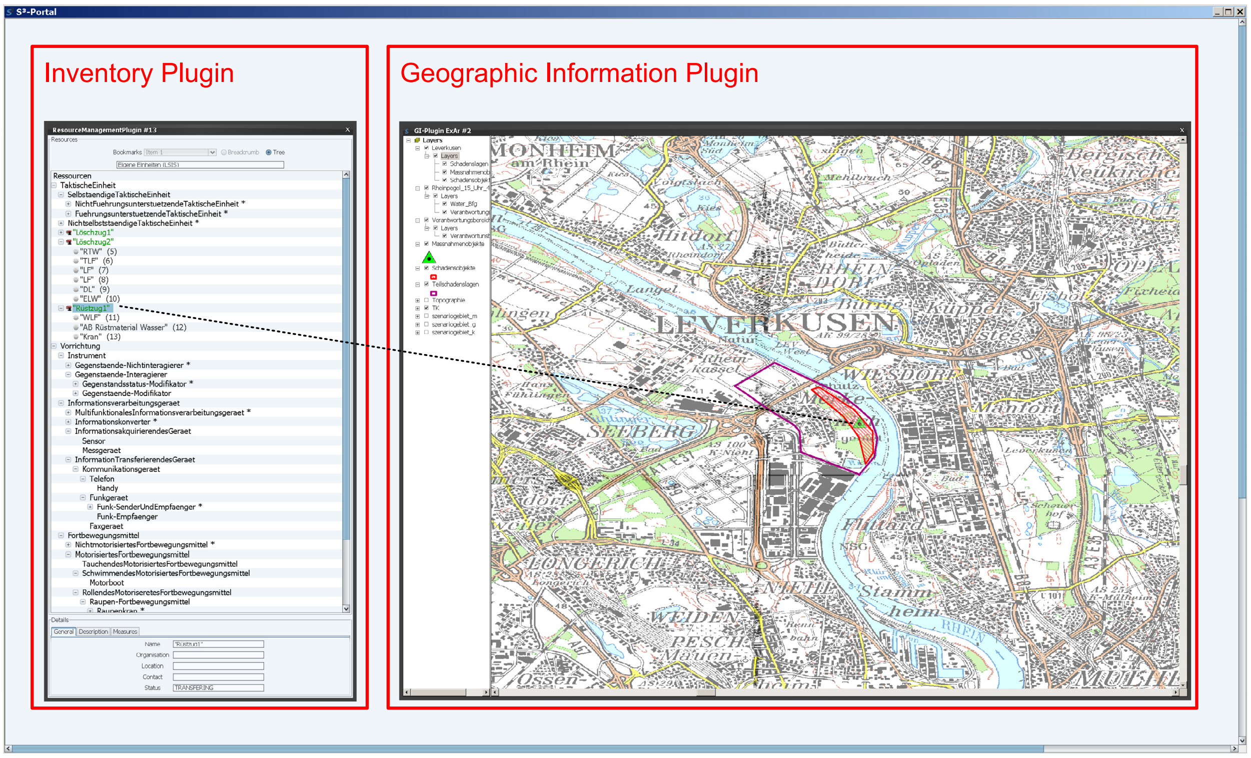

Figure 6: Screenshot of two plugins. The left-hand side shows an inventory plugin with a list of devices and material, the right-hand side a geographic information plugin presenting a spatial view. The two plugins are connected by a common interaction (depicted by the dotted line): selecting an object in the inventory plugin highlights that object on the map.

2. annotate the Java classes representing the objects that are exchanged between the plugins by establishing mappings between the classes and the real world domain ontology, and
3. define integration rules for cross-application interactions.

In the application ontology, the interactive map component is defined as being a part of the geographic information application[1]:

$$\begin{aligned} \forall c:\ & gi\#MapComponent\,(c) \\ & \quad \rightarrow \quad ui\#InteractiveComponent\,(c)\,. \qquad (1) \\ \forall c,p:\ & gi\#MapComponent(c) \wedge gi\#GIPlugin(p) \\ & \quad \rightarrow \quad ui\#hasSubComponent(p,c). \qquad (2) \end{aligned}$$

These definitions are required for the first step, i.e. defining the application ontology. The second step is to annotate the Java classes used to represent the objects involved in this interaction with the appropriate concepts defined in the domain ontology. This step has to be undertaken once for each plugin involved. Fig. 7 shows the mappings for classes and their attributes from the two plugins involved in the example. Those mappings are stored in the annotation registry (see Fig. 4).

As shown in Fig. 7, one mapping for a class and two mappings for attributes have to be defined for for the GI application. These mappings are represented by the following tuples, which are stored in the annotation repository:

[1]We use predicate logic notation with the following namespace conventions: *ui#* denotes the user interfaces and interactions domain ontology, *dom#* the real world domain ontology, *gi#* the geographic information application's ontology, and *dolce#* marks the imports from DOLCE. The domain ontology concepts used in these definitions are shown in Fig. 3 and 5.

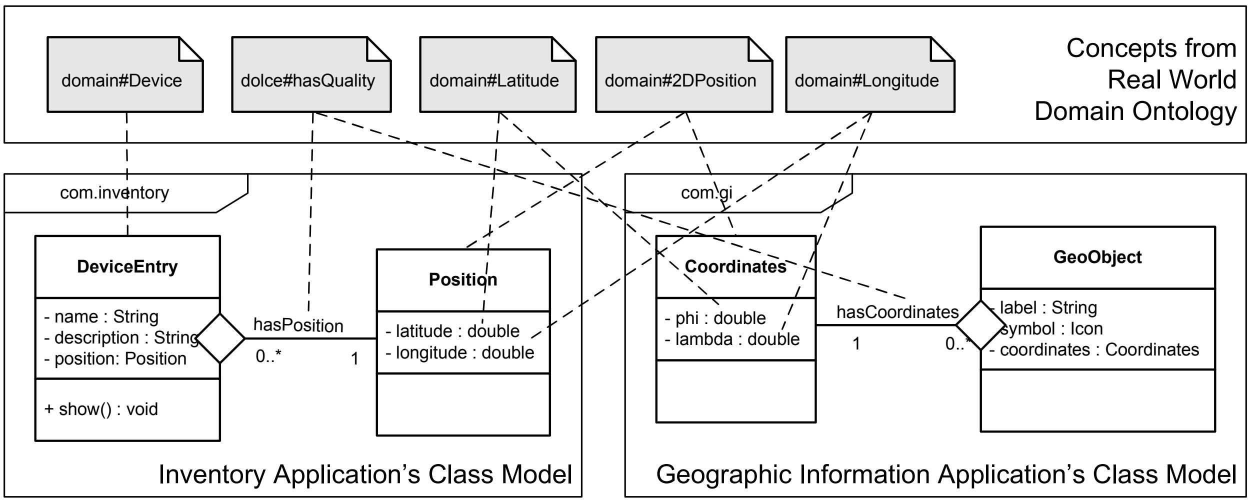

Figure 7: A Java class model annotated with the domain ontology

`com.gi.Coordinates`	–	`domain#2DPosition`
`com.gi.Coordinates`	`phi`	`domain#Latitude`
`com.gi.Coordinates`	`lambda`	`domain#Longitude`

The last step is defining the integration rule, i.e. the trigger and effect of a cross-application interaction, the condition under which it can be performed, and the component that supports the interaction[2]:

$$\begin{aligned}
\forall c, t, e, io_1, io_2, do : \quad & ui\#SelectAction\,(t) \wedge gi\#MapComponent\,(c) \\
& \wedge\, ui\#performedWith\,(t, c) \wedge ui\#involves\,(t, io_1) \\
& \wedge\, ui\#InformationObject\,(io_1) \wedge ui\#represents\,(io_1, do) \\
& \wedge\, ui\#InformationObject\,(io_2) \wedge ui\#represents\,(io_2, do) \\
& \wedge\, (dom\#Material\,(do) \vee dom\#Device\,(do) \vee dom\#Clothes\,(do)) \\
\rightarrow \quad & \exists i, e : ui\#Interaction\,(i) \wedge ui\#HighlightAction\,(e) \\
& \wedge\, ui\#supports\,(app, i) \wedge ui\#involves\,(e, io_2) \\
& \wedge\, ui\#hasTrigger\,(i, e) \wedge ui\#hasEffect\,(i, e)
\end{aligned}$$

However, this rule definition is not optimal. We have explicitly enumerated all the object categories (Material, Device etc.) that this interaction is feasible for. Furthermore, each time new plugins are integrated which support new categories of objects that this interaction would make sense for (such as a plugin showing a list of buildings),

[2]As many rule processing systems, such as OntoBroker, do not support rules with existential quantifiers in the head, such rules have to be skolemized [39] for further processing.

the interaction definition would have to be adjusted. Therefore, we discard the above definition and replace it with an expression that works with everything that has a position:

$$\begin{aligned}
\forall c, t, e, io_1, io_2, do : \quad & ui\#SelectAction\,(t) \land gi\#MapComponent\,(c) \\
& \land\ ui\#performedWith\,(t, c) \land ui\#involves\,(t, io_1) \\
& \land\ ui\#InformationObject\,(io_1) \land ui\#represents\,(io_1, do) \\
& \land\ ui\#InformationObject\,(io_2) \land ui\#represents\,(io_2, do) \\
& \land\ dolce\#hasQuality\,(r, q) \land dom\#2DPosition\,(q) \\
\rightarrow \quad & \exists i, e : ui\#Interaction\,(i) \land ui\#HighlightAction\,(e) \\
& \land\ ui\#supports\,(app, i) \land ui\#involves\,(e, io_2) \\
& \land\ ui\#hasTrigger\,(i, e) \land ui\#hasEffect\,(i, e) \qquad (3)
\end{aligned}$$

This improved definition supports every applications in which objects having a position can be selected. Furthermore, it will still be valid when the real world domain ontology is extended such that new categories of objects having a position are added, if new applications processing those objects are included, or if the inventory application is enhanced or even exchanged for a different one. Note that in the latter case, our approach provides means for evolving the integrated system as well as the real world domain ontology.

4.3. Execution

After these two steps have been performed, the pieces can be put together. Upon system startup, the domain ontologies as well as the application ontologies of all integrated applications installed are loaded by the reasoning component. The statements (1)-(3) formulated above thus become known to the reasoner.

For each application which is actually instantiated in a plugin, instance data for the application and its components as well as the information objects processed is needed. Such instance data can be written into the reasoner's A-box with a pushing approach, or dynamically gathered from the plugins during a query with a pulling approach. In [40], we have shown that only the pulling approach scales up for a larger number of integrated applications. Therefore, we use the pulling approach. When the GI application is started, the temporary instances (marked with the prefix *tmp#*) which can be pulled by the reasoner are:

$$gi\#GIPlugin\,(tmp\#p_1) \qquad (4)$$

$$gi\#MapComponent\,(tmp\#mc_1) \qquad (5)$$

More temporary instances are dynamically added when an action is performed. In the example, the user selects a device object in the inventory plugin. A select event is broadcast, containing the select event. The reasoner, which is connected to the event bus, reads the event and the object(s) contained therein, and inserts the corresponding temporary axioms:

$$app\#Select\,(tmp\#s_1) \qquad (6)$$

$$app\#InformationObject\,(tmp\#o_1) \qquad (7)$$

$$app\#involves\,(tmp\#s_1, tmp\#o_1) \qquad (8)$$

The detailed data about the information object can again be retrieved from the plugin using a pulling approach. For creating axioms from the information object (which is a Java object, technically), the mappings from the annotation repository are used. The resulting axioms are:

$$dom\#Device\,(tmp\#r_1) \qquad (9)$$

$$app\#represents\,(tmp\#o_1, tmp\#r_1) \qquad (10)$$

$$dom\#Position\,(tmp\#r_2) \qquad (11)$$

$$dolce\#hasQuality\,(tmp\#r_1, tmp\#r_2) \qquad (12)$$

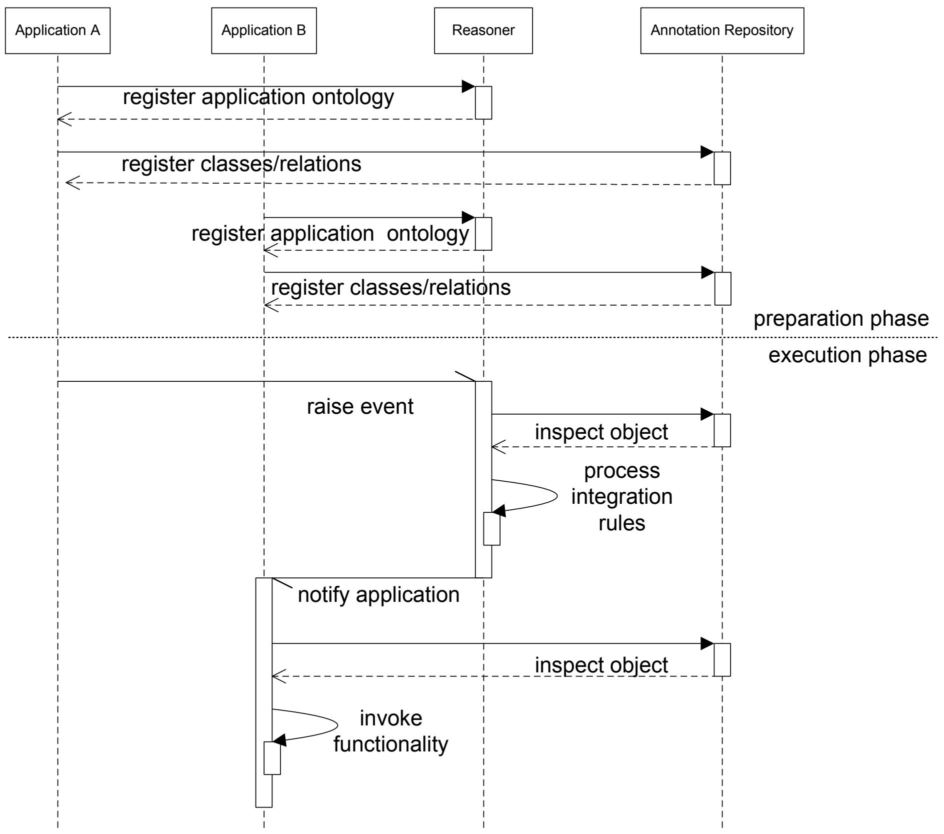

Figure 8: Execution of event processing by using a central reasoner. Events are raised by applications, sent to the reasoner asynchronously for processing. The reasoner uses the annotation repository to inspect objects contained in the event and processes integration rules for computing reactions to the event. The respective plugins are then notified and may again use the annotation repository to decode their notificiations.

Once these instance data have been added to the ontology, the reasoner can be queried for the interactions that can be triggered by the action (query variables are marked with ?) and the components that support the interaction:

$$\begin{aligned} &app\#Interaction(?I) \land app\#triggeredBy(tmp\#s_1) \\ \land\quad &app\#triggers(?E) \land app\#involves(?E, ?O) \\ \land\quad &app\#represents(?O, ?R) \\ \land\quad &app\#InteractiveComponent(?C) \land app\#supports(?C, ?I) \\ \land\quad &app\#Application(?A) \land app\#managedBy(?C, ?A) \\ \land\quad &app\#Plugin(?P) \land app\#encapsulates(?P, ?A) \end{aligned}$$

This query can be evaluated by the reasoner using the axioms stated above, thus retrieving a result list containing the interactions ($?I$) that can be triggered by this event, the triggered effects ($?E$), and the component ($?C$) and plugin ($?A$) supporting this interaction, as well as the plugin ($?P$) encapsulating the application. One of the results is the interaction of the geographic information application as defined above. The reasoner component can now use that information to create an appropriate event and send it to the geographic information plugin Fig. 8 shows the whole process. After that step, the temporary axioms 6 to 8 are removed.

It is noteworthy that the geographic information plugin gets an event which contains the original Java object produced by the inventory plugin. Since the geographic information application cannot process this object directly, its plugin has to, like the reasoner, query the annotation repository to analyze the object and retrieve the required information from that object using Java's reflection mechanisms.

This example has shown how axioms from the ontologies play together to facilitate the integration of user interfaces encapsulated in plugins, and how the reasoner can serve as an indirection for decoupling the integrated applications.

There are other interactions that can be treated by our framework similarly. One example is dragging and dropping objects, where the interactions defined also allow highlighting and possible drop locations and augmenting them with tool tips. Another example is the dynamic population of tool bars and menus [37].

Although a query to the reasoner is included in the event processing chain, our evaluations [40] show that even for a large number of integrated applications, the total event processing time is still way below the upper limit of two seconds given by the HCI literature [41, 42], since OntoBroker provides a lot of mechanisms for fine-tuning query performance, rewriting queries, etc. Details on those optimizations are out of scope of this paper.

5. Related Work

As pointed out in section 2, a lot of research has been conducted on using ontologies for application integration on the data source and on the business logic level. Database integration is a field where ontologies have been fruitfully applied [6]. On the business logic layer, the most prominent integration approach are semantic web services. Current research in this area, amongst others, aims at optimizing the matching of requirements and service descriptions and at automatically coupling several simple services for solving complex tasks [43, 44, 45].

On the contrary, little work has been done so far on using ontologies for application integration on the user interface level. There are, however, some works that use ontologies in the most popular user interface integration approaches, i.e. portals and mashups [46]. The approach described in [47] aims at using semantic web service descriptions for assembling portal applications from portlets. The work described in [48] and [49] shows how semantic annotation of contents delivered by portlets can be used to allow basic interactions between portlets. The work described in [50] shows how ontologies can help building mashup applications to integrate contents from diverse data sources in one mashup. Unlike our approach, which is rather interaction-centric, those approaches concentrate more on data integration and do not address the challenge of implementing cross-component interaction.

Some work exists in which semantic descriptions of software components (in general, not necessarily user interface components) are proposed. This work most often aims at building repositories of semantic descriptions for easing component-based software development. Software components are enriched with descriptions of their functionality as well as with non-functional properties, such as licensing policies. A software architect can then search those repositories with ontology-based queries [51, 52, 53]. The approach described in [54] uses ontology reasoning to find a set of compatible components for a given set of requirements. The main difference of these works compared with the framework described in this paper is that semantics are used at development time for finding appropriate components, not for assembling them at run time of the system.

While we concentrate on the use of ontologies for *integrating* existing user interfaces, other approaches, like the ones described in [55] and [56], use them for *generating* new user interface, thus providing a model-based approach. The latter work proposes the use of modular ontologies, comparable to the ones described in this paper. Other works employing ontologies in different fields of user interface development encompass requirements engineering [57, 58], configuration of user interface components [59], and migration between UI toolkits [60]. In contrast to our approach, all these approaches use ontologies at design-time, not at run-time of the software system.

6. Conclusion and Future Work

In this paper, we have shown an approach for ontology-based integration on the user interface level. We have presented a framework for developing and running plugins encapsulating applications, each of them using its own data source, business logic, and user interface. As an indirection between applications, the framework uses run-time reasoning to determine possible cross-application interactions, based on integration rules, so that no application has to directly react to other application's actions. We have shown that by employing that indirection and well-designed integration rules, systems built with our framework remain modular and can be easily evolved, e.g. by adding new categories of objects or new applications. Thus, our approach addresses one of the core problems of application integration on the user interface level, which is the implementation of cross-application interactions while preserving a modular and easily maintainable system, and without having to deeply understand each application to be integrated.

The framework foresees the use of a real world domain ontology which is not tightly coupled to the framework. Thus, the framework itself is domain-independent; by using an appropriate ontology, the framework can be used for building systems for arbitrary real world domains. Such universality is only possible by clearly separating the system description from that of the real world objects. Furthermore, we have conducted experiments showing that the framework scales up to a large number of integrated applications [40].

In the SoKNOS project [37], a large project from the emergency management domain, the framework described in this paper has been successfully used for integrating a running system from 25 applications developed by teams in eight different locations. The interactions between those applications are based on *application* ontologies, a shared *user interface and interactions* domain ontology and shared *real world domain* ontology.

Currently, we only consider pairwise interaction between applications. In future work, we aim at investigating the extension towards more complex interactions, such as use cases involving three or more applications, or system actions requiring user intervention. Furthermore, the state of applications has not been captured in the user interfaces and interactions domain ontology so far. Nevertheless, it might be helpful for defining interactions that can only be performed in certain situations. Therefore, extending the framework such that each plugin may expose information about its state will be part of our future work.

Another possible extension is the incorporation of an ontology for users and users' rights, such as described in [61]. With the help of such an ontology, the application developers could not only specify the interactions that an application supports, but also the conditions under which a user can perform those interactions. Next to users' rights, their preferences might also be modeled in ontologies and taken into account when determining interactions [62, 63, 64].

Furthermore, we have only considered information systems so far. In an information system, the user can only manipulate information objects. If we step from *information* to *control* systems, e.g. for automatically issuing orders or controlling mobile drones, it will be possible to manipulate domain objects *and* information objects. By separating both types of objects in our application ontology, we have already paved the way to enhance our framework in that direction.

The application ontologies may also be used to generate help for the user. To a small extent, this has already been implemented in our framework, such as highlighting possible drop locations when dragging an object [40], but there is the potential of driving this approach further. Verbalizing the plugin ontologies would be a consequent step towards generating help, which could be extended to to developing a component which can also answer users' questions, such as "how can I achieve X"?

Concerning the annotation of class models and their mapping to an ontology, we are currently restricted to $1:1$ mappings, i.e. mapping one class or attribute to exactly one concept in the ontology. There may be cases where such a mapping is not possible, since the class model and the ontology are too heterogeneous. To this end, we are currently developing a more sophisticated mapping mechanism [65].

At the moment, the developer has to write the application ontologies by hand, as well as the integration rules. Tool support such as a domain-specific ontology creation tool would further simplify the process of application integration.

In summary, we have presented an approach and a prototype implementation using ontologies for system integration on the user interface level. Thus, the reuse of user interfaces and faster integration of existing software is facilitated. Our approach shows how the idea of ontologies-based integration can be lifted to the user interface level, and thus carries the semantic web idea some steps further into the engineering of large, complex software applications.

Acknowledgements

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

References

[1] P. Naur, B. Randell, Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO (1968).

[2] W. Hasselbring, Information System Integration, Communications of the ACM 43 (6) (2000) 32–38.

[3] M. Fowler, Patterns of Enterprise Application Architecture, Addison Wesley, 2003.
[4] E. G. Nilsson, E. K. Nordhagen, G. Oftedal, Aspects of systems integration, in: ISCI '90: Proceedings of the first international conference on systems integration on Systems integration '90, IEEE Press, Piscataway, NJ, USA, 1990, pp. 434–443.
[5] F. Daniel, J. Yu, B. Benatallah, F. Casati, M. Matera, R. Saint-Paul, Understanding UI Integration: A Survey of Problems, Technologies, and Opportunities, IEEE Internet Computing 11 (3) (2007) 59–66. doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2007.74.
[6] A. Doan, A. Y. Halevy, Semantic Integration Research in the Database Community: A Brief Survey, AI Magazine 26 (1) (2005) 83–94.
[7] R. Studer, S. Grimm, A. Abecker (Eds.), Semantic Web Services - Concepts, Technologies and Applications, Springer, 2007.
[8] K. P. Sycara, M. Paolucci, Ontologies in Agent Architectures, 1st Edition, Springer, 2004, Ch. 17, pp. 343–364.
[9] B. A. Myers, M. B. Rosson, Survey on user interface programming, in: CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM, New York, NY, USA, 1992, pp. 195–202.
[10] N. Guarino, P. Giaretta, Ontologies and Knowledge Bases: Towards a Terminological Clarification, in: N. J. I. Mars (Ed.), Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing, IOS Press, Amsterdam, 1995, pp. 25–32.
[11] N. Guarino (Ed.), Formal Ontology and Information Systems, IOS Press, 1998.
[12] A. Gómez-Pérez, M. Fernández-López, O. Corcho, Ontological Engineering, Advanced Information and Knowledge Processing, Springer, 2004.
[13] W3C, RDF Vocabulary Description Language 1.0: RDF Schema, `http://www.w3.org/TR/rdf-schema/` (2004).
[14] W3C, OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/ (2004).
[15] J. Angele, G. Lausen, Ontologies in F-Logic, in: Staab and Studer [66], Ch. 3, pp. 45–70.
[16] J. Cardoso, The Semantic Web Vision: Where Are We?, IEEE Intelligent Systems 22 (5) (2007) 84–88. doi:http://dx.doi.org/10.1109/MIS.2007.97.
[17] G. Antoniou, C. V. Damásio, B. Grosof, I. Horrocks, M. Kifer, J. Maluszynski, P. F. Patel-Schneider, Combining Rules and Ontologies. A survey, Deliverable I3-D3, REWERSE (2005).
[18] D. Gasevic, N. Kaviani, M. Milanovic, Ontologies and Software Engineering, in: Staab and Studer [66], Ch. 28, pp. 593–616.
[19] F. Ruiz, J. R. Hilera, Using Ontologies in Software Engineering and Technology, Ch. 2, in: Calero et al. [67], pp. 49–102.
[20] G. Dobson, P. Sawyer, Revisiting Ontology-Based Requirements Engineering in the age of the Semantic Web, in: International Seminar on Dependable Requirements Engineering of Computerised Systems at NPPs, Institute for Energy Technology (IFE), Halden, 2006.
[21] N. Anquetil, K. M. de Oliveira, M. G. Dias, Software Maintenance Ontology, Ch. 5, in: Calero et al. [67], pp. 153–174.
[22] H.-J. Happel, S. Seedorf, Applications of Ontologies in Software Engineering, in: Workshop on Semantic Web Enabled Software Engineering (SWESE) on the 5th International Semantic Web Conference (ISWC 2006), Athens, Georgia, November 5-9, 2006, 2006.
[23] C. Puleston, B. Parsia, J. Cunningham, A. Rector, Integrating Object-Oriented and Ontological Representations: A Case Study in Java and OWL, in: Sheth et al. [68].
[24] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: Implementing the Semantic Web Recommendations, in: S. I. Feldman, M. Uretsky, M. Najork, C. E. Wills (Eds.), Proceedings of the 13th international conference on World Wide Web - Alternate Track Papers & Posters, ACM, 2004, pp. 74–83.
[25] S. Bechhofer, R. Volz, P. W. Lord, Cooking the Semantic Web with the OWL API, no. 2870 in LNCS, Springer, 2003, pp. 659–675.
[26] R. Chatley, S. Eisenbach, J. Magee, Modelling a framework for plugins, in: Specification and verification of component-based systems, September 2003, 2003.
URL `http://pubs.doc.ic.ac.uk/ModellingPluginFramework/`
[27] H. Paulheim, Ontology-based Modularization of User Interfaces, in: G. Calvary, T. C. N. Graham, P. Gray (Eds.), Proceedings of The 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS), ACM, 2009, pp. 23–28.
[28] D. Birsan, On plug-ins and extensible architectures, ACM Queue 3 (2) (2005) 40–46.
[29] E. Foundation, Eclipse.org home, `http://www.eclipse.org/` (2009).
[30] C. Wege, Portal Server Technology, IEEE Internet Computing 6 (3) (2002) 73–77. doi:http://dx.doi.org/10.1109/MIC.2002.1003134.
[31] J. Yu, B. Benatallah, F. Casati, F. Daniel, Understanding Mashup Development, IEEE Internet Computing 12 (5) (2008) 44–52.
[32] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, M. Matera, A framework for rapid integration of presentation components, in: Williamson et al. [69], pp. 923–932. doi:http://doi.acm.org/10.1145/1242572.1242697.
[33] ontoPrise, OntoBroker Website, `http://www.ontoprise.de/de/en/home/products/ontobroker.html` (2009).
[34] H. Paulheim, Ontologies for User Interface Integration, in: A. Bernstein, D. R. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2009, Vol. 5823 of LNCS, Springer, 2009, pp. 973–981.
[35] G. Babitski, F. Probst, J. Hoffmann, D. Oberle, Ontology Design for Information Integration in Catastrophy Management, in: Proceedings of the 4th International Workshop on Applications of Semantic Technologies (AST'09), 2009.
[36] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, WonderWeb Deliverable D18 – Ontology Library (final), Tech. rep., Laboratory For Applied Ontology, Trento, Italien, `http://wonderweb.semanticweb.org/deliverables/documents/D18.pdf` (2003).
[37] H. Paulheim, S. Döweling, K. Tso-Sutter, F. Probst, T. Ziegert, Improving Usability of Integrated Emergency Response Systems: The SoKNOS Approach, in: Proceedings "39. Jahrestagung der Gesellschaft für Informatik e.V. (GI) - Informatik 2009", Vol. 154 of LNI, 2009, pp. 1435–1449.
[38] S. G. Eick, G. J. Wills, High Interaction Graphics, European Journal of Operational Research 84 (1995) 445–459.
[39] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Pearson Education, 2003.
[40] H. Paulheim, Efficient Semantic Event Processing: Lessons Learned in User Interface Integration, in: Aroyo et al. [70], pp. 60–74.
[41] R. B. Miller, Response time in man-computer conversational transactions, in: AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I, ACM, New York, NY, USA, 1968, pp. 267–277. doi:http://doi.acm.org/10.1145/1476589.1476628.
[42] B. Shneiderman, Response Time and Display Rate in Human Performance with Computers, ACM Computing Surveys 16 (3) (1984) 265–285. doi:http://doi.acm.org/10.1145/2514.2517.

[43] S. Agarwal, R. Studer, Automatic Matchmaking of Web Services, in: International Conference on Web Services (ICWS'06), IEEE Computer Society, 2006.
URL \url{http://www.aifb.uni-karlsruhe.de/WBS/sag/papers/Agarwal_Studer-AutomaticMatchmakingOfWebServices.pdf}

[44] D. Roman, M. Kifer, Semantic Web Service Choreography: Contracting and Enactment, in: Sheth et al. [68], pp. 550–566.

[45] U. Küster, B. König-Ries, Measures for Benchmarking Semantic Web Service Matchmaking Correctness, in: Aroyo et al. [70].

[46] H. Lausen, Y. Ding, M. Stollberg, D. Fensel, R. L. Hernández, S.-K. Han, Semantic Web Portals - State of the Art Survey, Journal of Knowledge Management 9 (5) (2005) 40–49.

[47] T. Dettborn, B. König-Ries, M. Welsch, Using Semantics in Portal Development, in: Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, 2008, pp. 109–110.

[48] O. Díaz, J. Iturrioz, A. Irastorza, Improving portlet interoperability through deep annotation, in: WWW '05: Proceedings of the 14th international conference on World Wide Web, ACM, New York, NY, USA, 2005, pp. 372–381. doi:http://doi.acm.org/10.1145/1060745.1060801.

[49] L. Xianming, L. W. W. Chen, Research on the Portlet Semantic Interoperability Architecture, in: WCSE '09: Proceedings of the 2009 WRI World Congress on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2009, pp. 453–456. doi:http://dx.doi.org/10.1109/WCSE.2009.339.

[50] A. Ankolekar, M. Krötzsch, T. Tran, D. Vrandecic, The Two Cultures: Mashing Up Web 2.0 and the Semantic Web, in: Williamson et al. [69], pp. 825–834. doi:http://doi.acm.org/10.1145/1242572.1242684.

[51] H.-J. Happel, A. Korthaus, S. Seedorf, P. Tomczyk, KOntoR: An Ontology-enabled Approach to Software Reuse, in: K. Zhang, G. Spanoudakis, G. Visaggio (Eds.), Proceedings of the Eighteenth International Conference on Software Engineering & Knowledge Engineering (SEKE), 2006, pp. 349–354.

[52] P. Graubmann, M. Roshchin, Semantic Annotation of Software Components, in: EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, IEEE Computer Society, Washington, DC, USA, 2006, pp. 46–53. doi:http://dx.doi.org/10.1109/EUROMICRO.2006.54.

[53] D. Song, W. Liu, Y. He, K. He, Ontology Application in Software Component Registry to Achieve Semantic Interoperability, in: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC) - Volume II, IEEE Computer Society, Washington, DC, USA, 2005, pp. 181–186. doi:http://dx.doi.org/10.1109/ITCC.2005.216.

[54] O. Hartig, M. Kost, J.-C. Freytag, Automatic Component Selection with Semantic Technologies, 2008.

[55] B. Liu, H. Chen, W. He, Deriving User Interface from Ontologies: A Model-Based Approach, in: ICTAI '05: Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society, Washington, DC, USA, 2005, pp. 254–259.

[56] K. A. Sergevich, G. V. Viktorovna, From an Ontology-Oriented Approach Conception to User Interface Development, International Journal "Information Theories and Applications" 10 (1) (2003) 89–98.

[57] E. Furtado, J. J. V. Furtado, W. B. Silva, D. W. T. Rodrigues, L. da Silva Taddeo, Q. Limbourg, J. Vanderdonckt, An Ontology-Based Method for Universal Design of User Interfaces, in: Task Models and Diagrams For User Interface Design (TAMODIA 2002), 2002.

[58] K. Sousa, Model-Driven Approach for User Interface - Business Alignment, in: Proceedings of The 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'2009), ACM Press, 2009, pp. 325–328.

[59] P. Pohjalainen, Self-configuring User Interface Components, in: A. Dix, T. Hussein, S. Lukosch, J. Ziegler (Eds.), Proceedings of the First Workshop on Semantic Models for Adaptive Interactive Systems (SEMAIS), 2010.

[60] M. M. Moore, S. Rugaber, P. Seaver, Knowledge-Based User Interface Migration, in: ICSM '94: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, 1994, pp. 72–79.

[61] L. Kagal, T. Finin, A. Joshi, A policy language for a pervasive computing environment, in: Proceedings of the IEEE 4th International Workshop on Policies for Distributed Systems and Networks., 2003.

[62] S. Karim, A. M. Tjoa, Towards the Use of Ontologies for Improving User Interaction for People with Special Needs, in: K. Miesenberger, J. Klaus, W. L. Zagler, A. I. Karshmer (Eds.), ICCHP, Vol. 4061 of Lecture Notes in Computer Science, Springer, 2006, pp. 77–84.

[63] P. Korpipää, J. Häkkilä, J. Kela, S. Ronkainen, I. Känsälä, Utilising context ontology in mobile device application personalisation, in: MUM '04: Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia, ACM, New York, NY, USA, 2004, pp. 133–140. doi:http://doi.acm.org/10.1145/1052380.1052399.

[64] K.-U. Schmidt, J. Dörflinger, T. Rahmani, M. Sahbi, L. S. S. M. Thomas, An User Interface Adaptation Architecture for Rich Internet Applications, in: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (Eds.), The Semantic Web: Research and Applications. Proceedings of the 5th European Semantic Web Conference, ESWC 2008., no. 5021 in LNCS, 2008, pp. 736–750.

[65] H. Paulheim, F. Probst, Pragmatic Class Models Meet the Semantic Web - How dynamic, non-intrusive semantic annotation can build a bridge between the two worlds, currently under review.

[66] S. Staab, R. Studer (Eds.), Handbook on Ontologies, 2nd Edition, International Handbooks on Information Systems, Springer, 2009.

[67] C. Calero, F. Ruiz, M. Piattini (Eds.), Ontologies for Software Engineering and Software Technology, Springer, 2006.

[68] A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings, Vol. 5318 of LNCS, Springer, 2008.

[69] C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, P. J. Shenoy (Eds.), Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007, ACM, 2007.

[70] L. Aroyo, G. Antoniou, E. Hyvönen, A. ten Teije, H. Stuckenschmidt, L. Cabral, T. Tudorache (Eds.), The Semantic Web: Research and Applications, Part II, Vol. 6089 of LNCS, Springer, 2010.