Automatic Classification of Semantic User Interface Services

Károly Tilly
Oracle Hungary
Lechner Ödön fasor 7.
1095 Budapest, Hungary
+36705065195
karoly.tilly@oracle.com

Zoltán Porkoláb
Eötvös Lóránd University
Pázmány Péter sétány 1/C
1117 Budapest, Hungary
+36309225731
gsd@inf.elte.hu

ABSTRACT
Current user interfaces are ad hoc, application dependent and constantly change while offering the same functionalities in many different ways. This article investigates methods for creating semantic user interfaces, which are much easier to develop, learn, teach and use. The basic idea of semantic user interfaces is to analyze specific application domains (like word processing, file handling or application deployment), organize domain concepts into ontologies, associate user interface presentation attributes (like icons, menu labels and line mode command) to ontology nodes, and to use the ontology as a central controlling entity of application development and execution. The ontology is used inside a service oriented semantic user interface framework, whose elements and potential benefits are also explained.

The main contribution of this article is to investigate methods for analyzing and classifying computer system services, as a fundamental step of making the presented semantic user interface architecture operational. The problems and steps of service analysis are described and an automatic classification algorithm is presented based on formal semantic specifications and graph isomorphism. Implementation details and practical experiences are also outlined.

Categories and Subject Descriptors
H.1.2 [**User/Machine Systems**] – Human factors, I.5.3 [**Clustering**] - Algorithms

General Terms
Algorithms, Human Factors.

Keywords
semantic user interface; service oriented architecture; formal semantic specification; classification algorithm;

1. INTRODUCTION
State of the art graphical user interfaces (GUIs) are based on metaphores of real world objects and their related operations, which are well known to anyone from everyday life. Metaphores are presented by the user interface in graphical form as windows, icons and menus. Fundamental concepts of graphical user interfaces have remained basically unchanged since the eighties of the last century, when they were introduced in Xerox Star [18].

Graphical user interfaces are *intuitive*, which means that users are supposed to be able to associate a semantic meaning to the graphical (icon or menu) representations of the real world metaphores when they see them for the first time. Intuitivity admits *learning by exploring*, so that in theory no additional user training is needed to apply a GUI.

On the other hand at the state of the art there are no strict standards or known theoretical background, which would limit the designers of intuitive graphical interfaces in choosing arbitrary attributes (e.g. form of icons, text of menu labels or menu topology) for displaying user interface metaphors. That is why user interfaces become diverse even inside a specific application domain.

Diverse user interfaces make it almost impossible to teach novice users a sound background which they could apply in a generic way for a long time. While grammar school students receive generic knowledge in physics or biology, they receive short term, specific knowledge in informatics, just because there is no well known, widely accapted specification of computer system services and their common representations.

The main problem with creating and learning state of the art user interfaces is that the semantics of real world metaphores is associated to their user interface representations in an implicit way. We think that intuitivity is a very good thing from the user's perspective, but it is bad at the user interface designer's side. Certain parts of user interfaces, like icons, menu labels and menu structure along with the semantic specification of elementary services should be invariant, application, platform and device independent, while other properties, like skin or layout design of the interface may arbitrarily change. This approach is similar to the user interface of modern cars: they all have the steering wheel, pedals and important controls arranged the same way, while the shape of the body, form and colours of the seats and the dashboard may arbitrarily change. It guarantees that we do not have to learn to drive our new car again, while we may still find its interior and body design exciting.

This article investigates methods for creating user interfaces, which are much easier to develop, learn, teach and use. The basic idea is to analyze specific application domains (like word processing, file handling or application deployment), to organize domain concepts into ontologies, to associate user interface

SPLASH 2010, Workshop on Ontology Driven Software Engineering, October 18, Reno, Nevada, USA.

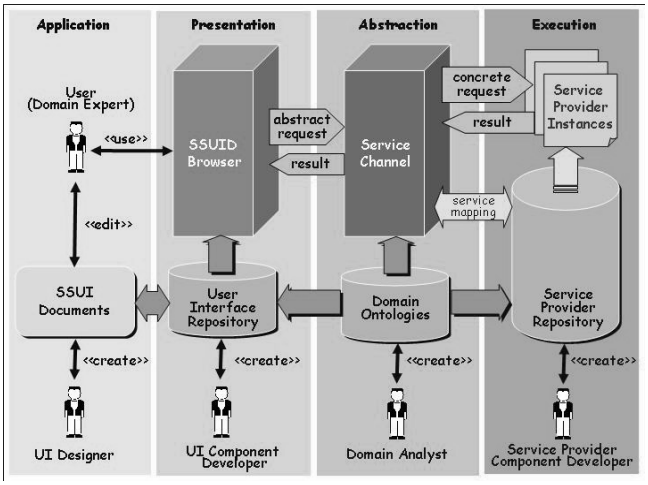

Figure 1. Semantic user interface architecture

presentation attributes (like icons, menu labels and line mode commands) to ontology nodes, and to use the ontology as a central control repository of application development and execution.

In Section 2 the architecture and potential benefits of ontology driven semantic user interfaces are outlined. The proposed semantic user interface architecture is service oriented, so services form the central part of the driving ontology. A fundamental step of making the architecture operational is to analyze and classify computer system services, which is the main contribution of this article. Section 3 explains methods for describing and classifying semantic user interface services. In section 4 semantic service specifications are introduced as our method for describing user interface services. Section 5 explains properties and generation of equivalent service graphs and outlines a classification algorithm for semantic service specifications based on graph isomorphism. Section 6 describes implementation details, while section 7 summarizes practical experiences.

2. SEMANTIC USER INTERFACES [1]

Semantic user interfaces are based on static, user editable documents (so called semantic user interface documents - SUIDs), which can contain references to arbitrary services, whose contracts are specified in domain ontologies.

According to Figure 1, SUIDs empower domain experts to create applications by editing sets of static documents, while IT professionals design and implement reusable user interface components and service provider components based on standard specifications stored in domain ontologies. To achieve this, domain ontologies must be created and an execution infrastructure is required.

The operation of semantic user interfaces follows a service oriented approach [16]. There has been recent efforts made for extending service oriented architecture with semantic elements including ontologies and refined functionalities, among others discovery, composition or mediation based on formal languages and automated reasoning [7,17]. While the operating environment of semantic service oriented systems is the internet containing an arbitrary number of distributed components, semantic user interfaces apply similar paradigms in smaller scale domains, like a single computer or a smart phone. Similarly, in service oriented architectures a service is generally considered a large, complex entity, while in semantic user interfaces it makes sense using small, elementary services organized into rigorous service ontologies.

Semantic User Interface (SUI) applications work in a *request/response* style. A *request* is defined as a list of objects, where each object has an associated semantic meaning. One of the objects defines a *service*, and the others define a set of *arguments*. Requests are executed by service providers, which implement activities with predefined contracts (services associated with specific argument names and types) stored in domain ontologies.

Architectural elements and operation of a semantic user interface infrastructure are explained according to Figure 1.

The *SUI Document Browser* renders SUIDs containing objects with specific *content types* registered in domain ontologies.

User interaction through a semantic user interface is based on a simple interaction model, the *presentation-navigation-selection-activation* cycle. According to this model the user interface *presents* a set of *content objects*. The user assembles *requests* by *navigating* desired objects; *selecting* them; *assigning a semantic role* to each of them, and finally performing a *terminating gesture* to initiate request execution.

Terminating gestures, like pressing the *Enter* key, signal the SUID Browser that the user has assembled a request, which should be forwarded for execution. The system *executes* the request, and returns a result, which is combined by the SUID Browser with the actually displayed pieces of contents.

The *Service Channel* assures *semantic separation,* which means that domain concepts of the user's mental model are explicitly separated from user interface components and application components. The set of separated domain concepts allow a dynamic mapping between elements of user interface documents and underlying application components.

In a technical sense the *Service Channel* dynamically matches requests to service provider contracts according to the roles and content types of request objects. The Service Channel determines the actual provider contract from the request, retrieves the providers offering the given contract from the Provider Registry, and selects one of them to execute. In other words service requests are not wired to specific executable entities as in state of the art systems (even in application servers, which merely offer spatial separation), but they are bound during run time, before execution.

According to domain ontology specifications the user interface always presents a specific service the same way independently of its arguments. For example, if service *copy* has an associated shortcut *Ctrl+C*, it can be used to copy selected content to the clipboard (as usual), but it can also be used to copy a source to a target argument independently of the fact, whether the content type of the arguments are local file, URL or even two files with different formats (which requires implicit conversion). This way the user can formulate semantically similar requests the same way, while the execution of requests may need providers with drastically different complexity and resource requirements.

3. ANALYSIS AND REPRESENTATION OF SEMANTIC USER INTERFACE SERVICES

The goal of analyzing user interface services is to find a minimal, set of services, to organize them into an ontology and to standardize them by associating service attributes like specification, help text, icon, menu label, command mnemonic, keyboard shortcut or mouse gesture.

The use of a generic service ontology has the following advenatges:

- *User interface designers* can reference service ontology items in semantic user interface documents by simply decorating certain objects with relationships to ontology nodes.
- *Component developers* can implement service providers based upon standardized service specifications stored in the ontology.
- *Users* can learn (can be taught) the attributes and specification of generic services stored in the ontology.

These properties make service ontolgies the central driving entity of a semantic user interface framework. Service Ontologies eliminate user interface diversity and support easy creation, learning, teaching and use of new interfaces.

Building a user interface service ontology includes the following steps:

1. Gather semantic information about services of existing applications considering their *presentation* and *informal, written specifications*.
2. Determine relationships between service specifications and build a classification.
3. Eliminate equivalent services from the classification.
4. Give meaningful names for service classes generated by the classification algorithm. Associate specification, help text, icon, menu label, line mode command mnemonic, keyboard shortcut and mouse gesture to all service items, which were not eliminated in step 3.

There are thousands of services offered by commercial computer applications in everyday use with hundreds of thousands of potential relationships. It is not just time consuming but practically impossible to find all relevant relationships and equivalences between specific services, and to reduce their number to probably several hundreds. That is why the process must be at least partly automated.

Gathering semantic information about services (Step 1) can be based on software documentation, which could be considered as a corpus of text. In our case, however, text based automatic ontology generation methods using linguistic analysis, lexical proximity or statistical analysis [11] cannot be applied. It is mainly because our concepts are not defined by words in a text and their internal relationships, but by larger text fragments containing several sentences to several pages. State of the art lingustic methods are simply not precise enough to detect exact semantic similarities between concepts under such circumstances.

That is why Step 1 cannot be automated, so semantic specifications of individual services must be explicitly defined. To make specifications sound and easy to process we use first order logic statements, which proved to be useful from multiple aspects for building ontologies in the past.

Formal ontologies represent categories through axioms and definitions in the form of (mainly first order) logic statements [6]. Since formal representations admit automatic inference (like resolution), additional information can be deduced and even the ontology itself can be automatically extended. Notice that service ontologies are not formal, and we have no intention to make automatic inference above ontological service descriptions. We just need formal service specifications as a starting point for classification.

Another relevant association to our approach is Formal Concept Analysis [4,5], which supports organizing a set of notions into formal *concept lattices*, by formulating the same set of logical propositions with truth values *true* or *false* above all studied notions. If there are n notions and m propositions, a T table of n rows and m columns is created. Any cell $T(i,j)$ contains *1* whenever proposition j is valid for notion i, and the cell contains *0* otherwise. Neighboring cells holding *1s* can be united into *concepts*, and further to concept lattices. This approach is however not appropriate to us, because the applied propositional logic description does not admit the formulation of generic statements, so it results in extraordinarily large concept tables. Furthermore this approach can only handle relations between notions, but cannot handle relations between propositions efficiently, which is crucial for describing services referring to, and modifying a set of interrelated objects.

Building classifications and eliminating equivalent services (Steps 2 and 3) can be automated. This is a good news, since these are the activities, where relationships and equivalences must be detected, whose potential number is by magnitudes of order higher than the number of basic services. To find relationships and to eliminate equivalent services we use an algorithm based on graph isomorphism [8,9,10], because it is much easier and more efficient to match equivalent graph representations of formal service specifications then to perform direct text based matching of first order logic statements (for more details see section 5).

Graph matching based on formal, language independent concept definitions has been applied in the field of schema matching, which is among others related to finding similarities and upper level categories in existing ontologies [14]. For example the S-Match semantic matching algorithm [12] finds semantic similarities between ontological trees containing nodes labelled by concepts. As a preprocessing step original concept labels are automatically compiled into statements of an internal concept language based on first order logic. S-Match handles different semantic relations between concept nodes (equivalence, more/less general, mismatch or overlapping). S-Match computes semantic relations between pairs of tree node concept labels as matrices.

A solution close to our approach is the *Graphdiff* [13] generic tree matcher, which perfroms approximate matching of trees based on subgraph isomorphism. Tree nodes have *types* and edges have *distances*. Only nodes with the same type can match, and the strength of match is determined by distances of neighboring edges. Graphdiff computes *scores* to qualify the strength of tree match based upon the number of matching edges and the ratios of their associated distances. To enhance practical runtime efficiency Graphdiff applies geometric hashing and heuristics derived from a valence based model.

Step 4 (specifying meaningful names and associating attribute values) again cannot be automated, because at this point human decisions are needed.

In the following sections a set of notions will be defined to establish a method for describing formal semantic specifications of services. An algorithm called SONG (Semantic user interface ONtology Generator) is presented, which uses formal service specifications to generate hierarchical service classifications, also referred to as *service ontologies*. Concepts and operation of the algorithm are described, and its practical use is introduced through a simple example.

4. SEMANTIC SERVICE SPECIFICATION

Service semantics describe the meaning of a service to the user. To capture common properties of arbitrary services, the meaning of the service must be specified in a formal way, which is totally independent of the software system, usage, appearance and other syntactic features of services.

Users formulate requests and await results according to their mental model of services. The classification method should be based on a formal description of user's mental service models. According to our supposed interaction model, service execution from the user's point can be considered as a request/response pair. To formulate a request, the user must fulfill some *preconditions* (e.g. supply arguments, assure certain content types or system states). When the request is executed, a response and a set of additional *effects* are generated for the user. For example if a user wants to copy a file into another, he has to specify two filenames. The response is a message, whether the file copy succeeded, and the effect is a new file, which holds the contents of the source file. Fortunately the details of execution are totally unimportant to the user, so we have a good chance to describe service models in a simple and compact form.

According to Figure 2 a semantic service specification is a named entity with a *precondition* and an *effect part*, each of which contain a set of first order logic predicates. The set of potential predicates and their arguments are predefined and their number is kept at a minimum level.

As an example let us consider the semantic service specification of the Windows Copy service shown in the left hand side of Figure 3. This specification says that there is a service named Copy in Windows, which can be initiated by specifying two arguments named *a1* and *a2*, where the initial value of *a1* is *v1*. When the service is executed, a new object is created named *a2*, whose value is *v1* (i.e. the value of *a1*). Furthermore the value of *a2* is stored in a persistent location (i.e. in a file).

5. CLASSIFYING SEMANTIC SERVICE SPECIFICATIONS

As a result of analysis a separate semantic service specification is created for each computer system service using the approach described above.

```
service('<service_name>','<application_name>');
                                                    precondition;
                                {<predicate>(‘<argument1>’{,’<argumentn>’});}
                                                    effect;
 {<predicate>(‘<argument1>’{,’<argumentn>’});}
```

Figure 2. Semantic Service Specification syntax elements

The task of the classification algorithm is to compare pairs of semantic service specifications and to determine their largest intersections. It seems quite easy: take the textual form of two semantic service specifications, and find the common parts, which can be considered as a class of semantic service specifications. This approach is however surprisingly hard to implement, because:

- The text of two semantic service specifications can contain arbitrary predicates in arbitrary order, which does not admit easy matching by e.g. ordering predicates lexically and pruning common prefixes. Parsing techniques are required, which are not easy to implement even using sophisticated pattern matchers like regexp parsers or similar machinery.
- Predicates reference each others through arguments. To allow incremental addition of new semantic service specifications without any knowledge of other existing specifications, argument names are consistent only inside a specific semantic service specification. In other words argument x1 does not necessarily denote the same argument in two semantic service specifications, which means that two textually equivalent predicates may be semantically different, while two semantically equivalent predicates may be textually different. To solve this problem, argument names must be normalized, which requires additional parsing.

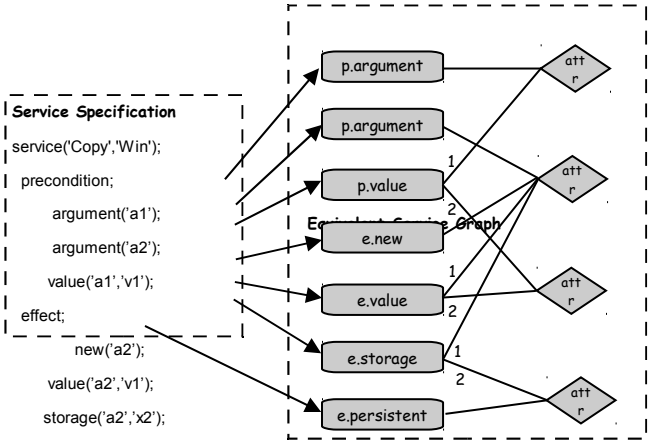

Figure 3. Mapping semantic service specifications to equivalent service graphs

.

These problems make direct textual matching clumsy. On the other hand it is easy to transform semantic specifications into an equivalent graph representation, where argument names are simply eliminated and relationships between predicates are expressed directly by adjacency of nodes inside the equivalent graphs. This way parsing and argument name normalization can be avoided, while a method is required for matching equivalent graph representations of semantic service specifications. This method is subgraph isomorphism, which generates the kind of results we need.

To see how semantic service specifications are transfomed into equivalent service graphs, let us consider the Windows Copy example in Figure 3. Equivalent service graphs have predicate nodes (one for each predicate in the original semantic service specification) and attribute nodes (one for each named attribute). Predicate nodes are labelled by the original predicate's name (e.g. *argument* or *value*) and the information, whether the predicate was part of the precodition or effect part (denoted by the *p.* or *e.* prefixes respectively). There is an edge between a predicate node and an attribute node whenever the

predicate contains a reference to the specific attribute. Edges are labelled by an integer number according to the position of the related attribute inside the original predicate. Notice that attribute names are neglected, and predicate-attribute references are transformed to adjacency relations of the predicate and attribute nodes inside the equivalent service graph. This solution guarantees that two service graphs can be matched even if the original semantic service specifications contain different attribute names.

Similarities between service specifications are discovered through detecting maximal isomorph subgraphs of any pairs of service graphs for a set of semantic service specifications. Simply speaking, two graphs are isomorph, if they can be redrawn in a way that they look exactly the same. A maximal isomorph intersection of two service graphs is either (1) equal to both service graphs, (2) a common subgraph of both service graphs or (3) empty.

In case (1) the two services are semantically equal, so one of them can be omitted from the classification. In case (2) the maximal isomorph intersection is a potential service class, which should be compared to other service graphs as well. In case (3) there is no semantic relationship between the two services.

The graph/subgraph relation between an equivalent service graph and its isomorph intersections defines a containment hierarchy, which is an ordering relation. This ordering relation is used as the basis of classification.

5.1 Service Classification Algorithm

We constructed an algorithm called SONG (Service Ontology Generator), which finds all maximal isomorph intersections of all potential pairs in a set of equivalent service graphs.

For the general case there is no graph isomorphism algorithm with worst case time complexity better than exponential [10]. Fortunately our service graphs have some nice properties, which admit solving the problem with low order polinomial practical time complexity algorithms.

1. Service graphs are always *bipartite*: Since first order logic predicates can never directly refer to another predicate, two predicate nodes or two attribute nodes cannot be adjacent.
2. Service graphs are relatively small. An average service graph has less than five predicate nodes, and our current experiences suggest that the number of predicate nodes will rarely exceed a dozen.
3. Node and edge labels of isomorph intersections must be equal.

Property (*1*) may look pretty, but in fact properties (*2*) and (*3*) prove to be much more useful. Property (*2*) lets us radically decrease the number of potential subgraphs to match, while property (*3*) allows drastical pruning of the search space while matching a single pair of graphs. For k service specifications having an average number of n predicates the estimated time complexity of our algorithm is between $O(nk)$ and $O(n^2k^2)$. Practical time complexity is nearer to $O(nk)$.

At the end service graphs are converted to OWL format to admit the last, manual step of building the service taxonomy. Attribute names are regenerated by simply assigning unique labels to each attribute nodes inside a service graph.

6. IMPLEMENTATION

A prototype of SONG has been implemented within the Oracle10g environment using SQL and PL/SQL. This choice may seem strange for the first sight, because relational databases are ment for solving data intensive and not computation intensive algorithmic tasks, so the resulting solution is supposed to have relatively low runtime efficiency. On the other hand it offers extraordinary flexibility at the implementation level, so this unusual environment is highly adventageous for prototyping. Underlying data structures could be designed easily using relational tables, while SQL is highly powerful for implementing complex graph matching operations in a simple and natural way. Furthermore Oracle 10g is optimized for efficient large scale parallel data manipulation, which admits high speed execution, if SQL operations are carefully tuned.

Experiments were started by creating semantic service specifications for a set of nearly one hundred Windows and FTP line mode system commands. Although this task requires precise domain knowledge, the work is quite straightforward. An average semantic service specification can be created within five minutes.

When semantic service specifications has been loaded, the taxonomy is generated by converting semantic service specifications to graphs and performing all steps of the service classification algorithm.

7. PRACTICAL EXPERIENCES

This section demonstrates the use of our method for classifying a small set of Ftp and Windows services. The examples are intentionally very simple, so that they can be easily followed.

We started from semantic service specifications for six Windows and Ftp services (*copy*, *restore*, *dir*, *get*, *ls* and *mdir*).

The resulting taxonomy is converted to an OWL text file to support manual editing.

Figure 4 shows the taxonomy opened for further editing using Protége 4 [15]. At this stage the service ontology must be completed through giving meaningful names for generated service classes, and associating specification, help text, icon, menu label, command mnemonic, keyboard shortcut and mouse gesture to all service nodes of the taxonomy according to step 4 of the classification process described in section 3.

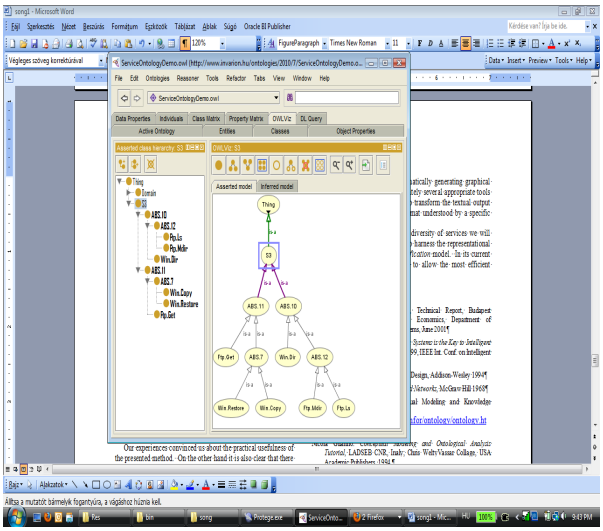

Figure 4. Editing classifications

Figure 4 shows, that according to the original semantic service specifications *copy* and *restore* are the same, and they have a strong relationship to the *get* service. An appropriate labeling for service classes *ABS.11* and *ABS.7* can be *Copy* and *LocalCopy* respectively. Similarly, *ls* and *mdir* are the same, and they are strongly related to the *dir* service. A possible labeling for *ABS.10* and *ABS.12* can be *Context* and *RemoteContext* respectively.

8. CONCLUSIONS

In this article a formal modeling method has been introduced for describing computer system services, and an automatic classification algorithm was outlined for generating service taxonomies. Practical applicability of the method was demonstrated through a prototype implementation and a simple example.

Our experiences convinced us about the practical usefulness of the presented method, which proved to be surprisingly efficient even using our current Oracle10g implementation aimed for prototyping. During our tests a taxonomy for 95 Windows and Ftp semantic service specifications could be generated well within 1 minute on a Lenovo T400 PC with 2.26 GHz dual Pentium P8400 CPU and 2 Gbytes of RAM. This implies that our current implementation is fully appropriate for generating service taxonomies of practical size as well.

The high run time efficiency was a positive surprise to us. After all Oracle's SQL engine is one of the best optimized software components in the world, which directly supports basic operations of the calssification algorithm (join and filtering), furthermore we carefully optimized the source code at the SQL-PL/SQL level.

On the other hand there is a lot of work ahead. To generate a taxonomy for a sufficient set of computer system services, their semantic service descriptions must be created, taxonomies must be generated, propely labelled and annotated.

The large number and potential diversity of services we will meet in the future may also harness the representational power of our semantic service specification model, especially in the number of required predicates.

A further important step is to extend our algorithm to automatic classification of other semantic user interface entities (like content types, roles and attributes). This way it can help us building a full featured semantic user interface ontology.

9. REFERENCES

[1] K. Tilly, Z. Porkoláb 2010: Semantic User Interfaces, *International Journal of Enterprise Information Systems*, 6, 1, (Jan-March 2010) 29-43

[2] G. Booch 1994: Object Oriented Analysis and Design, *Addison-Wesley* 1994

[3] G. Busacker, L. Saaty 1968: Finite Graphs and Networks, *McGraw Hill* 1968

[4] N. Guarino, C. Welty 2000: Ontological Analysis of Taxonomic Relationships, *Proc. of ER-2000 The International Conference on Conceptual Modeling* (October 2000)

[5] B. Ganter, R. Wille 1999: Formal Concept Analysis · Mathematical Foundations, *Springer Verlag* 1999

[6] J. F. Sowa: Ontology. *http://www.jfsowa.com/ontology/* (2003).

[7] D. Anicic, M. Brodie, J. de Bruijn, D. Fensel, T. Haselwanter, M. Hepp et. al. 2006: Semantically Enabled Service Oriented Architecture, *Technical Report, DERI, University of Inssbruck, Austria*

[8] J. Ullmann 1976: An Algorithm for Subgraph Isomorphism, *Journal of the ACM*, 23, (1976) 31-42

[9] K. Yamazaki, B. de Bodlaender 1997: Isomorphism for Graphs of Bounded Distance Width. *Algorithmica* 24, 2, (1997) 105-127

[10] S. Bachl 1999: Isomorphic Subgraphs, *Lecture Notes in Computer Science* 1731, (1999) 286-97

[11] C. Biemann 2005: Ontology Learning from Text: A Survey of Methods *LDV Forum* 20, 2, (2005) 75-93

[12] F. Giunchiglia, P. Shvaiko, M. Yatskevich 2004: S-Match: An Algorithm And An Implementation Of Semantic Matching, *Proceedings of the European Semantic Web Symposium*, *LNCS 3053*, (2004) 61-75

[13] K. Zhang, D. Shasha (1997): Approximate tree pattern matching, *in A. Apostolico, Z. Galil (eds): Pattern matching in strings, trees, and arrays. Oxford University Press*, (1997) 341–371

[14] E. Rahm, P. A. Bernstein 2001: A survey of approaches to automatic schema matching, *The VLDB Journal* 10, (2001) 334–350

[15] Protégé Ontolology Editor and Knowledge Acquisition System (2010), *Retrieved July 30, 2010 from Protégé Web Site*: http://protege.stanford.edu

[16] C. M. Mackenzie, K. Laskey, F. MacKabe, P. F. Brown, R. Metz 2006: Reference Model for Service Oriented Architecture, *OASIS Standard,* http://www.oasis-open.org/specs/#soa-rmv1.0, (2006)

[17] S. A. McIlraith, T. C. Son, H. Zeng 2001: Semantic Web Services, *IEEE Intelligent Systems*, 12, 2, (2001) 46-53

[18] Johnson, J., Roberts T. L., Verplank W., Smith D. C., Irby C., Beard M., Mackey K. 1989: The Xerox "Star": A Retrospective *IEEE Computer* 22, *9* (Sept. 1989), 11-26, 28-29