Designing User Interfaces through Ontological User Model

Functional Programming Approach

Syed K Shahzad
Knowledge Management Institute
Graz University of Technology
Graz, Austria
syed_khuram78@yahoo.com

Michael Granitzer
Know-Center Graz
Graz University of Technology
Graz, Austria
mgranitzer@tugraz.at

Klause Tochterman
Know-Center Graz
Graz University of Technology
Graz, Austria
klaus.tochtermann@tugraz.at

***Abstract*—Semantics and Ontological framework defines standards for concepts presented through computational models. This framework can also provide standards for the concept representation at User Interface level. A user model as the base of user interface can be built using formal ontologies. Language supporting Higher Order Functions can be used to build Graphical User Interfaces by mapping the user defined model with typed graphical user interface controls. This paper discusses a strategy to present the standard concepts of computational model to user interface through ontological user model.**

Keywords- Ontology; User Interface; Functional programming; Haskell; Universe of Discourse;

I. Introduction

A computer application is a computational model of a real world concept [1]. The concepts conceived by the user depend on the computational model. The computational model is built by human and represented concepts are based on the human understanding. These concepts are delivered to the user through the user interface of the computer application.

The Xerox Palo Alto Research Center [3] and IBM[1] presents "*The iceberg analogy of usability".* According to the aforementioned studies, the user interface is composed of three components 1) visual 2) interaction technique and 3) user model. The first two components collectively are visible parts of the Iceberg while the third component is invisible. The visible iceberg part is about look and feel of the user interface but represents 40% of the user interface. The major 60% part (user model) provides the concepts of computer application to the user. This underwater (hidden) part of iceberg includes the objects, properties and relations between them. The majority of recent work in user interfaces is focused on improving only the look and feel. Although many guidelines to improve the look and feel of a user interface have already been devised [11] yet no specific rules exist for modeling a user interface. On the other hand semantics and ontological frameworks are being used by many knowledge based systems to formalize the semantics of a domain via concepts and relationships. But in current computer applications, the same concepts are represented in many different forms on the visual component of a user interface and provide much displaced picture of the same concept. Hence, semantically similar concepts are perceived differently depending on the actual implementation of the user interface.

[1] http://www.ibm.com/developerworks/library/w-berry/

In order to consistently harmonize domain semantics with its visual representation requires standardization of the domain semantics, i.e. concepts and relationships, as well as their mapping onto standard visual components, so called widgets. Such an Ontological User Model (OUM), provides a direct mapping of the user model, component 3 of the iceberg, to the look and feel. Ontological user modeling can lower the cost of not only user interface development and maintenance but also for application testing. [3]

This paper discusses a strategy for representing and mapping ontological user model to a Graphical User Interface in the domain of spatial information systems. Spatial information system heavily rely on data represented as a 2D graph, like for example pathways, areas etc. Hence a 2D graph is discussed as an example implementation of GUI based on spatial ontology and its standardized visual representation. Based on related work, we formalize ontology for 2D graph and define a relational algebra in order to define manipulations and axioms (see Section II). We implement the relational algebra; using the functional programming language (Haskell) to create a user model, as in terms of the so called Universe of Discourse (UoD) (Section III). In a last step, shown in Section IV, we discuss the mapping of the Haskell implemented ontology (in the UoD) to GUI with a 2D graph example.

Within our work we show how to separate the user model and the according domain semantics from it visual representation and discuss how to generalize our findings for more complex domains (see Section V).

II. Ontology formalization

Specification and prototyping is essential for standardization of data and process modeling [5]. Ontology is an explicit specification of a conceptualization. Current research is exploring the use of formal ontologies for specifying content specific agreements for a variety of knowledge-sharing activities [8]. In our example ontological formalization specifies the conceptual model for a 2D graph as an example of ontology. Spatial ontology formalization is needed to get the specification of the conceptual model. The formal ontology can be then represented in relational algebra.

A. *Formal Ontology*

Good ontology and good modelling can be achieved in a specific domain through a precise representation of entities that exist in reality [10]. Here this domain is narrowed down to the discipline of spatial ontology for a 2D Graph model.

Formal Ontology can be intended as a theory of priori distinction among the entities of the world and among the meta-level categories [7]. Entities represented here are a 2D Graph, and Node and Edges as a part of graph. Every entity has an identity function as the identification for that entity.

Formal ontology deals with the categories and relations which appear in all domains and which are in principle applicable to reality under any perspective [6]. We state the entities and relation among them.

Our domain model is given as follows: Graph is called by a Title here referred as Graph name. The graph consists of some points and connection between them referred as Node and Edges respectively. Each node represents a location in 2D space and Edge represents the connection between two nodes. It is a straightforward definition of a graph to formalize entities and relationship among these entities. These specifications are stated using algebraic specification.

An example of simple 2D Graph:

Graph Properties

Graph	Graph ID		
	Graph Name		
		Node	Node ID
			Number
			Position
		Edge	Edge ID
			First Node
			Second Node

Table 1: Ontology Description of a Graph

Table1 present the properties and structure of a graph

This is an example of 2D graph just to describe how axioms and categories can be formalized using relational algebra and functional programming language. It is not any standard ontology for a graph.

Figure1 present an example instance of graph representation.

Example2D Graph

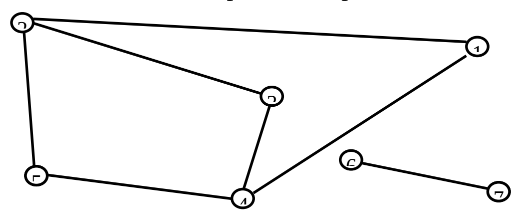


Figure1: An Example representation of Graph

B. *Relational Algebra*

The term "algebraic specification" originally referred to the use of algebras to model programs, and to use equatorial axioms to write specifications [2]. Here spatial ontology is formalised using formal ontology and relational algebra. Algebra represents spatial categories and relations in space dimension. Relations also defined potential functions of any object from a specific category.

Axioms can be specified as property functions for any object of a category X like:

```
class X
     p1(x) -> q1
     p2(x) -> q2
     ..
pn(x) -> qn
where
     p = property
     q = property value
     x = Object of Class X
```

Relations for 2DGraph graph examples are:

Graph contains a Name, List of Node and List of Edges. Graph is identified by GraphID

```
id :: Graph → GraphID
graphName :: Graph → Name
graphEdges :: Graph → [Edge]
graphNodes :: Graph → [Node]
```

Here Node contains a 2D location and a Number. Node identified by NodeID

```
id :: Node → NodeID
nodeNumber :: Node →  Number
nodePostition :: Node → Position
```

and Edge is a connections between two nodes. Each Edge contains a pair of Node referred by their NodeID while Edge is identified by EdgeID

```
id :: Edge → EdgeID
startNode :: Edge → ID
endNode  ::Edge → ID
```

Potential Operations on Graphs also provide relations in the entities for the graph. Relational algebra can also define potential processes.

```
class X
     $f_1(x)$ -> $s_1$
     $f_2(x)$ -> $s_2$
     ..
     $f_n(x)$ -> $s_n$
where
     f = Function
     s = Behavior of function
     x = Object of Class X
```

C. *GUI States*

A user model represents a state at the user interface. This state is the picture that comes up by the current properties measurements of concept instances. These states provide a concept of user model to GUI. Here we will discuss different states of Graphical User Interface. Any potential process can update the measurement/values of the properties resulting in state update. Initiation of any potential process of the concept reads the current state of the user interface and responds accordingly by updating the state. Potential processes for this example provide following relations:

```
   newGraph ::Name → [Node] → [Edge] → S(GUI )
   deleteGraph ::GraphID  → Database → S(GUI)
   updateName ::GraphID → Name → Database → S(GUI)
   updateGraph :: (GraphID → Graph ) → S(GUI)
```

Any node and Edge function can update a state of Graph being a part of Graph. Graph object will propagate the signal of change of state to GUI.

```
   addNode :: Number → Postion → S(Graph)
   deleteNode :: NodeID → S(Graph)
   updateNodeNumber :: NodeID → Number → S(Graph)
   updateNodePosition  ::  NodeID  →  Position  →
S(Graph)
   addEdge :: NodeID → NodeID → Graph → S(Graph)
   deleteEdge :: EdgeID → S(Graph)
   updateStartNode :: EdgeID → NodeID → S(Graph)
   updateEndNode :: EdgeID → NodeID → S(Graph)
```

The algebraic specification is implemented using functional programming language. Here Haskell is used for implementation for this relational algebra.

D. *Universe of Discourse (UoD)*

User Interaction in/out GUI through UoD

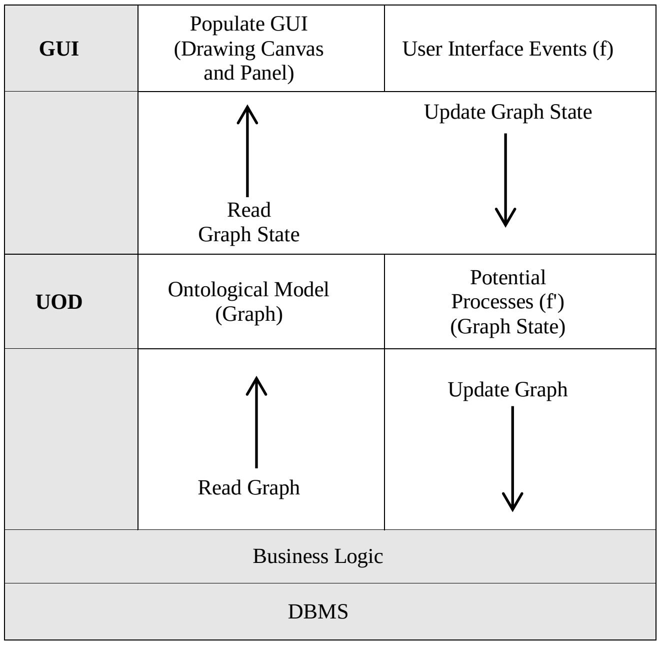

GUI
Populate GUI (Drawing Canvas and Panel)
User Interface Events (f)
Read Graph State
Update Graph State
UOD
Ontological Model (Graph)
Potential Processes (f') (Graph State)
Read Graph
Update Graph
Business Logic
DBMS

Table2: The mapping of User Model to GUI through Data Flow

Our work uses the idea of a Universe of Discourse. UoD is a representation of ontological structure of conceptual model which can be directly mapped to User interface. (REF : Frank's Draft for UoD)

UoD represents ontological user model. UoD works using the formalized ontology to represent the behavior of a computer application. It is then directly mapped to a GUI and forms the base model for a user interface. It is based on ontological definition in axioms defining properties and relations. It ensures the data values from or to the user interface are values fulfilling the ontological constraints.

In short, the UoD defines an ontological structure from an application and maps it to the GUI elements.

III. HASKELL APPROACH

A. *Why Haskell*

Haskell is an advanced purely functional programming language. It provides features like higher-order functions, lazy evaluation, equations and pattern matching, strong static typing and type inference, and data abstraction [15]. Hakell allows types (inferred and functional types) through axioms to form classes and category as a domain concept.

wxHaskell with a powerful second order language can be used to generate a user interface based on ontologies. A functional programming language allows the designer to examine whether the model is a correct representation of reality with respect to the ontology, and if the objectives and expectations are fulfilled [4]. Algebraic representation of ontology can be written down directly to Haskell. Representation of these relations can be done with a mapping function which can map functions (ontological functions) to a function (I/O function). In this paper Haskell is discussed for representation of relation algebra. Haskell represent relational algebra by its typed structure. Moreover State Monads facilitate for representation of states.

B. *Representing Algebra in Haskell*

Graph consists of nodes and connection between nodes referred as edge. Representation of Algebra is achieved in Haskell through

- Simple types t
- First order function types f::t → t'
- Higher order functions g :: f → f'

Here is data-structure implementation in Haskell for:

- Node

Nodes are defined as some points attached with an number. nodeNumber is an identity for the node and nodePosition defines the position of node in 2D space.

```
   data Node = Node {
                       nodeNumber :: Number
                     , nodePosition :: Position
                     } deriving (Eq, Ord, Show)
   where
   newtype NodeNumber = NodeNumber Int| NodeNumber
unknown
   type Position = Point
```

- Edge

Edges are defined with two nodes startNode and endNode

```
data Edge = Edge {
                   startNode :: Number
                 , endNode :: Number
                 } deriving (Eq, Ord, Show)
```

- Graph

Graph is container or aggregate for lists of nodes and edges with a name. graphName is an identity for the graph.

```
data Graph = Graph {
                     graphName :: Name
                   , nodeList :: [Node]
                   , edgeList :: [Edge]
                   }
where
type Name = String
```

C. *Manipulating States (State Monads)*

Business logic of software provides a flow of information and a structure to process and manipulate the values base on the concepts. The GUI provides information in measurements of properties to the user and operations for the user. Operations available at the user interface allow user to see and update the current measurements. Current values of instances of any concept form a picture a state at User use interface. Querying and updating operations thus result in reading and update the current state.

Haskell represents states using state monads [12].

```
   newtype State s a = State { runState :: (s →
(a, s)) }
```

where 's' is a state type and 'a' is an observation from the state as a simple type or a data structure of an object.

All observations are used to read any measurement at resulting state of state monad. The state is stored in shape of measurements in quantity or quality to data storage. State monads are used here to read and write the current state of Data Store to get and change state of Graph through ontological processes. The ontological processes are stated as relations in UoD.

Here states are used as a computer model of reality.

```
class classX a where
     f → State a objectX
```

Data Storage is instantiated for any class CategoryX. f is the function to update the current state or read an objectX.

e.g. manipulating DB states

```
     f → State DB object
```

The same way ontology is instantiated for UI Model

```
     f → State UoD object
```

Here any function can read or update the state of the database model or the user interface model. For 2D graph examples, categories are defined on the base of behavior. All potential processes (for Nodes, Edges and Graphs classes) result in the reading or the update in current state of graph. Haskell defines manipulation of state 'a' for mentioned graph processes as following:

```
   class a
   newGraph :: Name → State a GraphID
   deleteGraph :: GraphID → State a Bool
   graphbyName :: Name → State a (Maybe Graph)
   setGraphName :: GraphID → Name → State a Bool
   newNode :: Number → Position → Name → State a
NodeID
   deleteNode :: NodeID → Position → State a
NodeID
   updateNumber :: NodeID → Number → State a Bool
   updatePosition :: NodeID → Position → State a
Bool
   getNumber :: NodeID → State a Number
   getPosition :: NodeID → State a Position
   addEdge :: NodeID → NodeID → Name → State a
EdgeID
   deleteEdge :: EdgeID → State a EdgeID
   setStartNode :: EdgeID → Number → State a Bool
   setEndNode :: EdgeID → Nr → State a Bool
   edgebyNode :: NodeID → State a [Edge]
```

Type of an object to some state either changes in state or observation of current state.

D. *Data types and Typed Classes*

Objects keep properties and potential process collectively called behaviour of an object. Categories are made of the base of commonalities in behaviour as criteria of the class.

Using Haskell, a type is not only a simple type, there are also types defines using axioms. Ontological axioms can be used to specify the exact domain concept as a type of any object. Also functional type can be defined as a class to represent behavior of an object or category.

In our example of 2D Graph,

NodeNumber is a type which is associated with positive integers but is it neither an integer type nor a positive integer. NodeNumber cannot be added, subtracted or manipulated like integers. It have only two functions = or ≠. Haskell define these types with scales as functors.

NodeNumber = f (x)

where - x $\in Z^+$

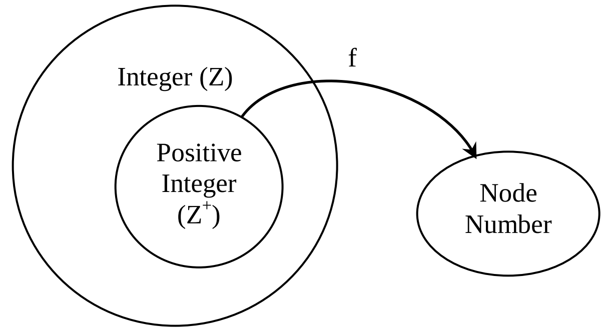


Figure2: Node Number as a Functional Types to represent ontology

Haskell wrap these types to manipulate it different from simple integer types.

IV. MAPPING DATA MODEL (UOD) TO GUI

Mapping ontology to GUI provides the concept of representation of meta-data and possible operations. These relations were stated in UoD. A transparent interface is the best interface where User interacts directly to the conceptual model entities and performs potential processes. GUI can have a bijective mapping to the formalized conceptual model. The bijective mapping create real picture of direct interaction to the conceptual model.

Here GUI shows the current state of Graph at drawing canvas and provides controls to initiate any potential process. Graph data structure provides meta-data of Graph. Haskell provides higher order functions to map ontology to GUI. We have used simple graph example, this graph can be any weighted or directed graph, or a more sophisticated applications like Shortest Path. For every different application, there are different concepts presented in axioms and different categories. The UoD provides the framework for any application and can be mapped to the GUI.

A. GUI construction tools

Different graphical libraries like Wxwidgets, OpenGL, QT and GTK available to plug with Haskell. Different GUI development libraries are also available based on above mentioned graphical libraries available. Here wxHaskell and wxGeneric is used for construction of GUI based on formalized ontology. WxGeneric allows building customised controls based on the data type.

B. Drawing at Canvas

Current state of reality for a 2D graph can be shown to user as 2D picture (*Figure3*). It is the ontological show method or the representation of vector values. This picture can be mapped to GUI by draw function. Draw function draw graph to a display screen. The graph structure represents Universe of Discourse (UoD). Drawing can visualize all objects from UoD at a specific scale. For example of 2D graph, the drawing is a graphical representation of 2D graph at specific scale.

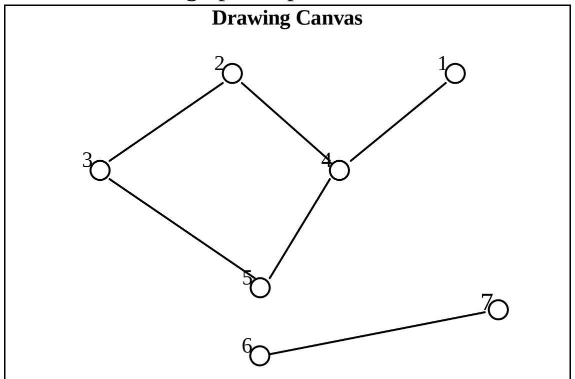


Figure3 : Application Interface for representing Graph

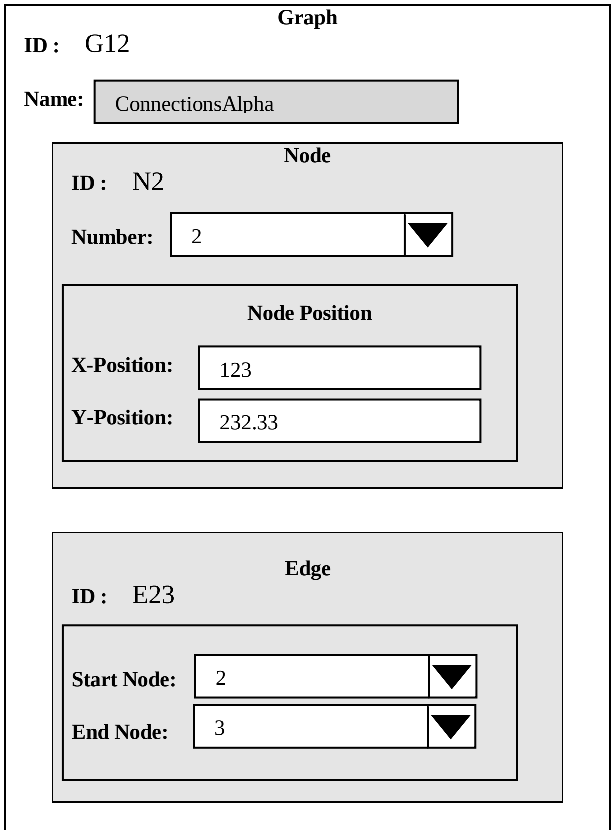
C. Composability and Typed User Interface

Argument here is: GUI should ensure the types and categories of. WxGeneric provide facility to make specific typed controls. Typed controls by wxGeneric represent the conceptual structure of categories to GUI. Panels can group simple type objects and these simple type objects can be represented with typed controls. These are composite controls like text box for Number type. [13] For 2D Graph example typed controls are used through composability

```
   type  IntEntry  =  Composite  (TextCtrl  ())  (IO
Int, Int →
      IO ())
   type NodeIDEntry = Composite (TextCtrl ()) (IO
NodeID, NodeID → IO ())
   type NodeNumberEntry = Composite (IntEntry ())
(IO NodeNumber, NodeNumber → IO ())
```

A typed panel can be created to manipulate a structured data type. Typed panel is composed of typed controls. Each simple type can be mapped to GUI through typed text boxes, list boxes, radio buttons, check boxes etc. The mapping function have domain of types and data structure in UoD and co-domain is the set of GUI data controls.

Current example creates a very simple interface panel, which is based on a user model. It ensures the consistency and standardization (*Figure4*).



ID : G12

Name: ConnectionsAlpha

ID : N2

Number: 2

X-Position: 123

Y-Position: 232.33

ID : E23

Start Node: 2

End Node: 3

Figure4: Panel to get and set data values of the graph properties through typed controls

D. Mapping Processes to GUI

Mapping of processes to GUI is more complex than mapping of data structure to panel or drawing. Processes results in Change of graph state. So it is a function type which needed to be mapped. Comparative to imperative programming styles, Haskell can provide a higher order mapping functions to map these processes. Still it is bijective morphism though mapping function is a higher order functions.

The mapping function has the domain of User Interface events and the co-domain is the set of processes in UoD. The processes can be mapped to the events of GUI controls. Implementation details of these mapping using different controls and clipping initiation of processes at the specific event is not discussed here. The majority of GUI designing guidelines and aim of transparent GUI can be achieved by these mapping functions.

Here the issue is with mapping processes having parameters; the GUI needs to provide some control to select objects to initiate any processes on them. Specific processes can be done to objects of specific category. In graph example the setGraphName can change name of a graph, and parameter here is the GraphID.

At any operation which requires a user input can be represented as a panel or dialogbox for getting input parameters. Graph example use notebooks for each class containing buttons for operation at each tab. At button click event panels appears for input data form user. This process input panel contains typed objects for every simple type or identity type for a data structure like node which is identified by the nodeID. Each process panel has an Ok button to run process in the UoD which is further mapped to Data Storage using Haskell State monads for the data store.

E. Limitations and constraints

There are many issues related to GUIs which are not discussed here like

- Synchronization of sequence of user actions with ontological processes.
- Data storage and buffering at UoD level.
- Database error handling.
- Selection of objects bounding domain of objects to perform few operations e.g. deleteNode process can be done only visible nodes at UoD. These constraints are not on category but at the set of values.

V. Conclusion

Spatial Ontologies can be mapped to GUI by a function using an intermediate layer of UoD. This layer works as a user model for a user interface. It provides the application semantics in form of an ontological framework of the concepts to be represented at the GUI. It provides relations to specify meta-data and potential processes for different categories. Mapping from UoD to GUI can be bijective to present an exact view of conceptual model at the GUI. In imperative style languages much translation and complex systems are involved to map from the business logic to the GUI. Higher order mapping functions can clip ontological processes to the GUI controls. Typed control can be used to read or provide objects of specific category for any potential process of the category. Any graphical control and its behavior can be mapped with the ontological definition of objects which includes the processes as a typed control. These controls can provide true picture of the concept.

VI. Future work

In this paper user action or sequence of actions are not discussed but for the next layer of ontological process is ontological definition of user action to perform certain tasks like clicking at canvas for selection of nodes or edges. The meta-data or ontology at User Interface should include ontological description to access and activate the possible process at any object (User-Interface Ontology).

Mapping of Temporal ontology with GUI is also needed to be discussed in future. This mapping will require synchronizing user generated event through IO devices with temporal ontological processes.

WxGeneric works on the data structure to construct an automated GUI. Where potential processes need to be implemented separately from data structure. There is still need of an abstract level GUI construction tool which can implement the typed controls with the classes. Where processes in classes can be handled as events handlers for events of UI ontology to update GUI state.

Acknowledgment

This contribution is partly funded by the Know-Center and the Higher Education Commission of Pakistan. The Know-Center is funded within the Austrian COMET program -- Competence Centers for Excellent Technologies -- under the auspices of the Austrian Federal Ministry of Transport, Innovation and Technology, the Austrian Federal Ministry of Economy, Family and Youth and the State of Styria. COMET is managed by the Austrian Research Promotion Agency FFG.

References

[1] A.U. Frank , "Draft An Empirical Ontology for GIS", V5 Sept. 2005.

[2] A Car and A.U. Frank "Formalization of conceptual models for GIS using Gofer," Computers, environment and urban systems. Vol. 19, no. 2, pp 89-98, March-April 1995.

[3] K. Alexander, G. Valeriya. From an ontology-oriented approach conception to user interface development. International Journal "Information Theories & Applications. Vol. 10, num.1, p. 87-94, 2003.

[4] L Cardelli and P Wegner, "On Understanding Types, Data Abstraction, and Polymorphism" Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

[5] A. U. Frank, and W. Kuhn. "Specifying open GIS with functional language", Lecture Notes in Computer Science, Volume 951, pp 184-195, 1995, doi 10.1007/3-540-60159-7.

[6] P. Grenon and B. Smith, "SNAP and SPAN: Prolegomenon to Geodynamic Ontology", Spatial Cognition and Computation,Vol 4: 1 pp 69–103, March 2004.

[7] N. Guarino, "Formal ontology, conceptual analysis and knowledge representation", International Journal of Human and Computer Studies, vol43(5/6): pp 625-640.1995

[8] T. R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge Sharing", International Journal of Human and Computer Studies, vol43(5/6): pp 907-928.1995

[9] W. Kuhn "An Image-Schematic Account of Spatial Categories", Lecture Notes in Computer Science, vol 4736, pp 152-168, August 2007, doi 10.1007/978-3-540-74788-8.

[10] B. Smith "Beyond Concepts:Ontology as Reality Representation", p. pp 73-84, FOIS 2004

[11] S. L. Smith and J. N. Mosier, "Guidelines for designing user interface software". Report MTR-10090, The MITRE Corporation, Bedford, Massachusetts, 1986.

[12] Haskell Documentation: Haskell Hierarchical Libraries (mtl package) http://cvs.haskell.org/Hugs/pages/libraries/mtl/Control-Monad-State.html

[13] M. Lindstorm, "Proposal: Adding composability to wxHaskell" 2008. Haskellville weblog,

http://lindstroem.wordpress.com/2008/03/16/proposal-adding-composability-to-wxhaskell/

[14] P.Hudak, "Conception, evolution, and application of functional programming languages"n ACM Computing Surveys (CSUR), vol 21, issue 3, pp359-411, September 1989.