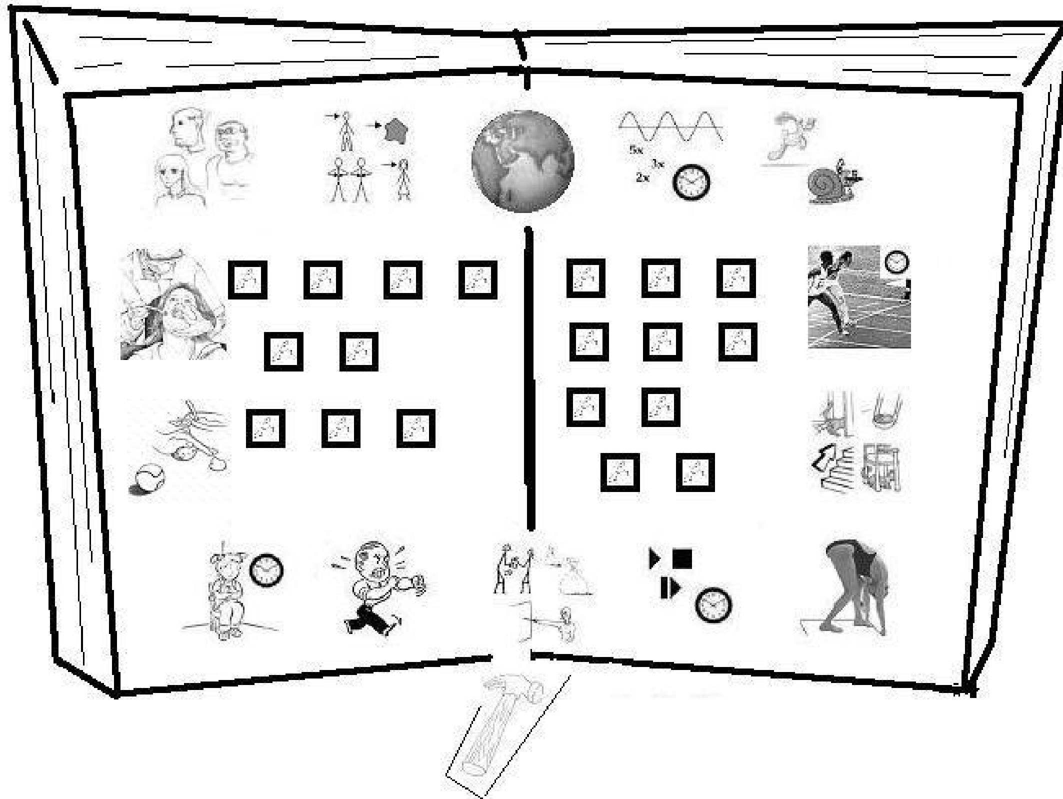
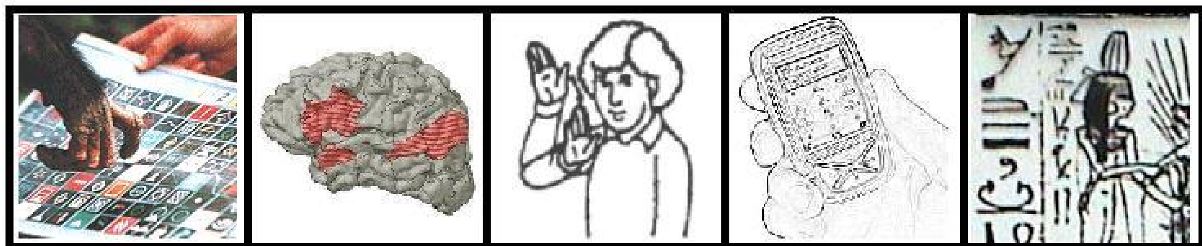
Visual Iconic Language

A Framework for Visual Iconic Languages





Anne C. van Rossum

June 2006

VilAug IconMessenger IconNet

TU Delft

Delft University of Technology

Visual Iconic Language: A Framework for Visual Iconic Languages

Graduation Committee:
Dr. drs. L.J.M. Rothkrantz (thesis supervisor)
Dr.Ir. C.A.P.G. van der Mast
Ir. F. Ververs
MSc. S. Fitrianie

June 2006

Anne C. van Rossum, student number 1006916
Master thesis, 30 June 2006,

Delft University of Technology
Faculty of Electrical Engineering,
Mathematics and Computer Sciences
Media & Knowledge Engineering
Man-Machine Interaction (MMI) Group

VilAug IconMessenger IconNet

Contents

1 Introduction **1**
1.1 Problem Setting & Background 1
1.2 Problem Description 2
1.3 Motivation 3
1.4 Project Description 3
1.4.1 Goal 4
1.4.2 Proposed Solution 5
1.4.3 Project Phases 5
1.5 Practical Applications 6
1.5.1 Emergency Center Scenario 6
1.5.2 Drive-In Scenario 6
1.6 Report Structure 8
1.7 Report Syntax 8

I Literature & Visual Language Study **11**

2 Literature Survey **15**
2.1 Background Theory 15
2.1.1 Cognitive Psychology 15
2.1.2 Linguistics 18
2.2 Applied Theory 22
2.2.1 Sign Languages 23
2.2.2 Ontology Engineering with Semiotics & Philosophy 24
2.2.3 Current Ontologies 28
2.3 The Visual Linguistic Field 32
2.3.1 Overview 32
2.3.2 VIL 34
2.3.3 Lingua 36

II Model & Architecture **39**

3 Ontological Model **43**
3.1 Model Description 43
3.1.1 Formal Definitions 44
3.1.2 Iconicon 45
3.1.3 Ontology 46

3.1.4 Grammar . . . 56
3.2 Applications & Examples of the Model . . . 58
3.2.1 Iconology . . . 58
3.2.2 Grammology . . . 64
3.2.3 Graph Mutations . . . 65
3.3 Model Requirements . . . 68
3.3.1 User Actions . . . 68
3.3.2 Data Structure . . . 68
3.3.3 Parsing Grammar . . . 68

4 Software Model **71**
4.1 Context . . . 71
4.1.1 Stakeholder Involvement Diagram . . . 71
4.1.2 System Overview . . . 72
4.1.3 Task Requirements to System Functionality . . . 72
4.2 Software Model . . . 77
4.2.1 Introduction . . . 77
4.2.2 IconMessenger Functionality . . . 77
4.2.3 IconNet Functionality . . . 78
4.2.4 VilAug & JUNG Extension Submodels . . . 78
4.3 Meta-Handling Data . . . 82
4.3.1 Factories . . . 83
4.3.2 Model-View-Controller . . . 84
4.3.3 Registry . . . 85
4.4 Representing Data . . . 85
4.4.1 Graph Representation . . . 86
4.4.2 Iconic Representation . . . 86
4.4.3 Verbal Representation . . . 88
4.5 Loading & Saving Data . . . 88
4.5.1 GraphML API . . . 89
4.5.2 GraphML Parsing . . . 90
4.5.3 File Modularity . . . 90
4.6 Performing Mutations . . . 92
4.6.1 Mutation Types . . . 92
4.6.2 Mutators . . . 93
4.7 Displaying the Graph . . . 94
4.8 Browsing the Graph . . . 94
4.8.1 Family Analog . . . 94
4.8.2 Different Node Types . . . 95
4.8.3 Time and Speed . . . 96
4.9 Filtering the Graph . . . 96
4.10 Parsing Grammar . . . 98
4.10.1 Earley Parser . . . 98
4.10.2 Grammar Rules API . . . 99
4.11 Peircean Functionality . . . 99

III Implementation 103

5 VilAug Implementation 107
5.1 Introduction . . . 107
5.2 Package Overview . . . 108
5.3 Implementation Guidelines . . . 109
5.3.1 How to Browse the Iconology . . . 109
5.3.2 How to Implement a User Action . . . 110
5.3.3 How to Display the Iconic Concepts . . . 113
5.3.4 How to Layout the Graph . . . 115
5.3.5 How to Show a new Dialog . . . 117
5.3.6 How to Run as an Applet or Application . . . 117
5.3.7 How to Enrich the Iconic Concepts with Additional Data 118
5.3.8 How to Store the Iconology in a Different Way . . . 120
5.3.9 How to Parse Grammar in a Different Way . . . 121
5.3.10 How to Add an Entire New Language . . . 122

6 Graphical Tour 125
6.1 IconMessenger GUI . . . 126
6.2 IconNet GUI . . . 127
6.3 User Actions . . . 127
6.3.1 Adding Concept to Visual Language . . . 127
6.3.2 Upgrading a Relation . . . 131

IV Evaluation 135

7 VilAug Evaluation 139
7.1 Model Comparison . . . 139
7.2 Model Scope . . . 141
7.3 Project Value . . . 142
7.4 Interface Design Analysis . . . 144
7.4.1 Content of Design Principles . . . 145
7.4.2 Time involved in Implementation . . . 145
7.5 Framework Analysis . . . 146
7.5.1 General Model Functionality . . . 146
7.5.2 Specific Model Functionality . . . 146
7.5.3 Specific Model Pecularities . . . 148
7.6 Recommendations . . . 148
7.6.1 Orthography Tools . . . 148
7.6.2 Semantic Restrictions . . . 148
7.6.3 Graphical Inflection . . . 149
7.6.4 Translation Tools . . . 149
7.6.5 Implementation Improvements . . . 150
7.6.6 Usability Assessment . . . 150

VilAug IconMessenger IconNet

8 Conclusion **151**
- 8.1 System 151
- 8.2 Comparison 152
- 8.3 Results 152
- 8.4 Future Directions 153

A Iconicity - More Dimensions **155**

B Categorization in VIL & Lingua **157**
- B.1 Modifiers in Fillmore's Case Grammar 157
- B.2 Primitive ACTs in Schank's Conceptual Dependency Theory . . 158
- B.3 Categorization of Verbs in VIL 158
- B.4 Categorization of Nouns in VIL 158
- B.5 Categorization of Nouns in Lingua 159

C Porting to VilAug **161**
- C.1 Porting from VIL and Lingua to VilAug 161

D Picture Credit **163**

E Class & Interface Hierarchy **165**

Bibliography **VIII**

Abstract

In this awesome era of overwhelming scientific discoveries and technological inventions, the computer can hold its own. It was in ancient times that mankind changed from using hieroglyphs and pictures to abstract symbols like letters. But now! The digital world does increase the rate at which we can manipulate pictures. And its global village character provides us with thousands of different languages. Is this the time that a new visual iconic language will see the light?

A visual iconic language uses sentences of concatenated icons to communicate. Certain combinations of icons are prescribed or prohibited. These restrictions form a grammar. In previous created visual languages case grammar theory by Fillmore and conceptual dependency theory by Schank are used. A grammar profits from formulations of its rules in the sense of icon categories or types. This leads to an ontology particular aimed to use icons instead of words as its basic building blocks.

In this thesis such an ontology, called an Iconology, is modelled. A demonstrator of this model is provided by the framework VilAug. Besides being an iconology stores the framework VilAug several visual languages and grammars. It also provides a graphical environment to actually create and alter visual languages and grammars. This environment is called IconNet, and comparable to a graphical WordNet environment. An additional user application, IconMessenger, is embedded in the form of an instant messenger that uses messages made of icons. The entire software package is a proof of concept in regard to the ontology and demonstrates that a new discipline of visual linguistics can be exploited in knowledge representation.

Preface

This report describes the research done in a graduation project at the Man-Machine Interaction group of the Delft University of Technology, headed by Dr. L.J.M Rothkrantz in fulfillment of my master thesis. Several investigations in the field of visual iconic languages have been done by researchers and master students of this group (Data and Knowledge Systems).

Project

The project started with a suggestion of my supervisor Léon Rothkrantz to investigate the work of Siska Fitrianie (and her predecessor Iulia Tatomir) who had implemented an application called Lingua. This application used visual iconic messages to convey information for tourists in a foreign country. An extensive literature survey had to be performed to investigate the advantages and disadvantages of such a visual iconic language. After this study I became convinced that it was possible to create such a visual iconic language, although not a universal one.

The work of Paul Leemans with his Visual Inter Lingua inspired me to start to design and construct a framework that combined the best of the applications of my predecessors (like Siska Fitrianie, Iulia Tatomir and others) at the Delft University. This application is labelled VilAug in honour of the language VIL. The results of the theoretical research assignment and the path to the final implemented application is written down in this very document.

Report overview

This report is divided into several parts. The first part is about theoretical background regarding visual iconic languages. This is a summary of the literature research [1] with much relevant, additional material. The second part of this report is dedicated to the task of explaining the model and the architecture of the framework that has to be built, and to describe the visual linguistic model that has to be implemented. The third part of this study investigates the implementation of this framework and becomes specific until the level of the programming language and the implemented visual languages. That framework is the application VilAug. This reports ends with a fourth part, that evaluates the framework in the sense of implemented functionality and a quick tour decorated by screenshots.

This report is in meant to be read by people of many backgrounds. The theory behind it is described in the first part (on page 11) and the theoretical model chapter (on page 43). The functionality of the framework is described

in the software model chapter (on page 71) and the evaluation chapter (on page 139). See also chapter with the graphical tour (on page 125). Details about implementation can be found in the third part (on page 103) and the Java documentation (javadoc) provided with the software. For a good introduction I would recommend to read the first and last part of this report.

This report is also written to accellerate further development of this application by the programming internet community. Therefore recommendations are at times spickled all over the place. The project is online at `http:\\sourceforge.net\projects\vilaug\`. A CD is compiled that contains previous applications, javadoc, the implemented framework, the visual languages it contains, theory and other material. A copy can be delivered upon request.

Another function is to guide iconic language creators in the future. With that in mind are assumptions, remarks and definitions explicitly described as such. I hope you will enjoy my work.

Acknowledgments

I want to thank the *developers* of the graphical package JUNG, Java progamming language, programming environment Eclipse, modelling environment ModelMaker, the [W3C] format XML and transformation language XSLT, the format GraphML, layout program GraphViz, typesetting system LaTeX, document processor Lyx and bibliography tool BibTeX, and all others that provided me with necessary tools.

I want to thank also people that kindly responded to my questions and posts on mailing lists, like the Conceptual Graph list, the SignWriting list and the JUNG Support List. And, I want to thank Siska Fitrianie and Paul Leemans in particular.

And everybody that played a positive role in my life I want to thank sincerely, especially my dear girlfriend. I want to thank Léon Rothkrantz for the conversations we had together.

Delft, June 2006

Chapter 1

Introduction

Can people communicate with pictures? Are pictures not much more ambiguous than words? How to implement a grammar for a language that uses pictures instead of words? These theoretical questions are important, but there are questions behind these questions. What is the difference between a word and a concept? What is the difference between an icon and a concept? How would it be to "speak" really only in icons? Can a model be created that uses only icons and no words? And a computer model? Which functions has such a model when it is implemented? How can the user profit from this application?

Section 1.1 describes the problem setting and problem background. Section 1.2 contains a brief problem description. Section 1.3 explains the motivation behind solving this particular problem. Section 1.4 describes the project, its goals and subgoals. Section 1.5 mentions two possible practical applications of visual languages. Secion 1.6 describes the structure of the report and section 1.7 introduces syntax used in this report.

1.1 Problem Setting & Background

The field of visual linguistics does not exist. There is no profession named "visual linguist". Yet, some people investigated matters beyond human language, beyond sounds, beyond words, beyond nouns and verbs. People like Charles Fillmore and Roger Schank, who studied semantic[1] elements at the basis of language. Several - although artificial - visual languages exist. Languages that not only use icons, but also have rules to combine them and to restrict certain combinations. One such a visual language is Visual Inter Lingua (VIL) created by Leemans. His dissertation [2] gives much background and theory about matters that play a role in the field of visual linguistics. Because the disipline of visual linguistics does not exist, will a definition of the visual linguist, as envisioned in this report, be given:

Definition 1. A *visual linguist* is a person who creates (and studies) visual iconic languages. Visual iconic languages contain concepts, icons, grammatical classes and grammatical rules. The visual linguist is like a teacher of physics

[1] meaningful

at a school in a Deaf community[2]. Such a teacher has to invent signs. When people start to use visual iconic languages on a bigger scale, a visual linguist observes their behaviour and the linguist's job becomes more descriptive than prescriptive.

The need for a visual iconic language has been felt for a long time. Especially a language that would enable to communicate on international scale. Many of such auxiliary languages have been developed, like Esperanto and Ido. The theory [1] indicates however that *universality* is probably *not* the goal to be strived for. Fortunately, it does stipulate other advantages that will be found in using a visual iconic language. So, developing a framework for visual languages won't be in vain. These theoretical issues are briefly discussed in Part I of this report.

The terms visual language and visual *iconic* language do all come down to the same idea. A language made out of icons or pictures. Iconic means *resemblance* in some way or another. A visual iconic language is more iconic than a natural spoken language. It may be the case that learning such language is easier. Non-speakers can intuitively grasp its meaning. However, a visual iconic language is not always magically self-explaining. This is also handled in Part I.

The composers of visual languages tackled a threefold problem. First had they to define the concepts and pictures of a language. Then a grammar had to be defined. A grammar combines icons in grammatically correct sentences. And last but not least a way to categorize them had to be invented. All of these steps should be based upon a sound theoretical basis. The composition of a visual language is an interesting problem, but the actual problem that is approached in this thesis is explained in the following paragraph.

1.2 Problem Description

There is a more urgent need than a new, theoretical sound, visual language. Several visual languages exist, although largely unknown. Each of them has its own set of icons and grammar rules. Each of them works in a specific domain. But, the creation of a visual iconic language and the application around it, were hereto heavily interwoven. However, computerizing a new exotic language and developing a word processor or instant messenger are completely different tasks! The developer of a visual iconic language does until now need much programming experience to compose her language. It would be better if the visual linguist would be relieved from that programming task.

The data from different types of visual languages should be captured in one "ontological" (see subsection 2.2.3) model. This model should be tailored to a representation of icons and the visual modality in general. A framework that knows how to store several iconic language may be seen as a demonstrator of this model. The framework should also be enriched with tools. One of these tools enables the visual linguist to perform tasks like adding icons to a language

[2] the term 'Deaf' and 'Deaf community' (with a capitalized D) will be used for all people, hearing and not hearing in a Deaf community who use a common means of communication, namely a sign language

and changing the grammar. Another tool is in the form of an instant messenger that uses icons[3].

So, the problem can be formulated as the question:

> PROBLEM DESCRIPTION: Can we create a theoretical model and corresponding software model to provide an environment for a visual linguist to create a new visual language: add icons, add grammatical icons, add grammar rules?

This is - very briefly - the topic of this thesis. The material at hand - pictures instead of text - asks at unsuspected times an entire other approach in regard to ontology engineering.

1.3 Motivation

The field of visual linguistics is very new. Not many people considered its theoretical depths and widths. No one ever proposed a framework that experts in this new field of visual linguistics are able to use without straying off in programming issues. This is a challenge!

Above that, visual linguistics leans itself for multidisciplinary knowledge. Findings from sign language can be used, as well as linguistics itself, semiotics, cognitive psychology, neurophysics and knowledge engineering. Because of this multidisciplinarity, many problems are seen in a new light. This paved the way for exploring a new model, a datastructure that can be seen as a generalized semantic web. This ontological model is used to store all the visual languages, its icons and the relationships between them. The software (model and implementation) benefits from this ontological model, but its actual functionality is based upon its own power and usability.

The idea to work at the borders of contemporary scientific fields is highly motivating.

1.4 Project Description

The goal of this project is to create a framework that provides an environment for (developing) visual iconic languages. A framework that can store different visual iconic languages. Languages that have their own dictionaries of icons, sets of grammar rules and icon classification structures. The framework should provide the creator of a visual iconic language with tools to append and delete icons to and from her language. It should be possible to write grammar rules with or without order. It should be possible to reuse icons from other languages in a new language. It should be possible to structure the icons of a visual language in a custom hierarchy. The focus of this research will be on developing such a general and flexible framework.

When developing such a framework it is important to check not only that the features of above are provided, but also to populate the framework with visual languages at hand. Two visual languages will be added to this application, namely the language VIL (by Leemans) and the language Lingua (by Fitrianie).

[3] like Lingua

No new visual language will be developed in this paper. The emphasis is upon providing a framework for visual iconic languages and no rigorously testing of a particular implementation will be performed. The incorporation of these two different languages is considered as a proof of concept. The visual language expressivity and application functionality will be evaluated.

1.4.1 Goal

The goals should be established precisely, to prevent disappointments. It is not the goal of the - to be developed - application to be the next tool in row that can handle visual iconic messages in a *specific domain*. The visual message strings that can be sent from firefighters to an emergency center (see subsection 1.5.1), the messages that a tourist can display on her PDA to the hotel owner, a menu with icons at a drive-in restaurant (see subsection 1.5.2), a touch panel that aids disabled people in commmunication, are "just" a few practical applications, that *show the power* of such an iconic language once it comes into existence.

It is neither the idea to develop a perfect, *universal language* (and accidently a visual one). Nor is it the intention to develop a universal grammar, or perfect data representation of iconic concepts.

It is the goal of this research to invest the characteristics of visual iconic languages, to study corresponding grammars and enable the wealth of this visual modality to be developed in full in an environment that fits this modality in particular. The model, the data representation, everything, has to be designed, such that *the communicative value of icons is captured*. This approach does probably impose specific restrictions or sheds new light upon certain common ontology characteristics that are unintentionally embedded. Characteristics that actually belong to the verbal domain (words, verbs or even sounds). The potential power of a visual iconic language is regarded to be comparable to a sign language.

This thesis assignment should result in an *application that accelerates the development of visual languages like the development of terminologies in sign language* (see figure on this page). The figure shows creation of physical terms in Norwegian sign language (by Roald [3]).

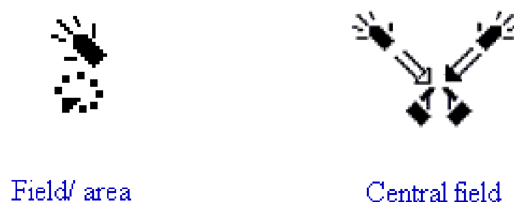


Figure 1.1: Physics Terminology Development

The software model orginating from the theoretical model will be implemented. The implemented framework should solve the problem as described in the problem description (see section 1.2). It will be rather a framework than a

VilAug IconMessenger IconNet

domain application. At the beginning of Part III can the requirements underlying this framework be found.

1.4.2 Proposed Solution

In this thesis is a model developed that uses icons instead of words, the Iconology. A software model and implementation is created, the framework VilAug. This model and this framework belong to a new field of *visual linguistics*. Linguistics that regards visual languages instead of spoken, written or signed languages. The framework will contain two tools. One tool to communicate visual messages with an instant messenger, the IconMessenger. Another tool, the graphical environment IconNet, to create and adapt visual languages.

1.4.3 Project Phases

The project can be divided in several phases. First the literature research assignment has to be undertaken. This study is very important because it does not only give us an overview of the endeavours of our predecessors, but it does also indicate what problems have to be solved in this area of visual linguistics, and it provides clues about possible solutions from many different fields of expertise. The existing visual languages deserve special attention.

This is followed by the phase of creating a proper model for the framework and ends with implementing this framework and populating it with at least two visual languages. Two types of model can be distinguished. A theoretical, ontological model that incorporates pecularities of visual linguists and a software model that provides a programming language independent overview of the software. In the last phase the results that are achieved will be evaluated.

Table 1.1: Project Phases

phase	topic	description
I	Literature & Visual Language Study	A thorough investigation of the field of visual linguistics, signaling problems, and providing advice for problem solving. This entails studying the languages / applications VIL (Leemans) as well as Lingua (Fitrianie).
II	Theoretical and Software Model	Design a theoretical and software model. Create UML descriptions for the framework to be developed.
III	Implementation	Implement the framework and import the languages VIL and Lingua with their grammars.
IV	Evaluation	Evaluate the framework about having the power to solve the situation described in the problem setting.

Not all these phases have to be completed sequentially, but it is probably convenient to maintain this order.

1.5 Practical Applications

What are exactly the practical and social consequences of a visual iconic language? What kind of applications can be invented? In what kind of situations would a visual iconic language provide solutions to certain problems?

1.5.1 Emergency Center Scenario

One such situation is the havoc generated by a large-scale disaster, that asks for cooperation between multiple service providers. Policemen, firemen, ambulance personnel, they would all profit from a central emergency center that guides them and receive additional information about the scope of the calamity. When the size of a disaster is gigantic, like a tsunami afflicting several countries, the emergency center should be able to convey messages quickly regardless of the mother tongue of the people involved. This can go in two directions. The rescue workers, health care workers, and others should be able to send messages to the emergency center. The center should be able to send messages to the victimes and other people nearby the epicenter of the disaster. At places where many nationalities can be found, like on airports, this can be crucial on a smaller scale too.

The crux is to have a kind of communication that is relatively easy to comprehend or easy to learn. The amount of ambiguous sentences should be not much larger than in spoken languages. The communication method should be sufficient complex to be able to convey the kind of messages that have to be conveyed in this kind of situations. Messages about fire, smoke, accidents, traffic jams, or about hurricane, flood, water level, should be part of the repertoire that can be used. It is even possible to use this information to construct a two-dimensional map of the area involved. Information about water levels can be used to induce which roads have become submerged fro example.

Such an easy way to communicate is possible by making use of visual iconic languages. Languages that use pictures, icons, to communicate. All people involved can communicate with the emergency center by entering visual messages. See figure 1.2 for an impression of such a scene. When a terrorist attack takes place, ambulance personnel can report the severity of injuries, doctors the amount of room for new patients in a hospital, policemen the situation on the street, firemen the situation in a burning or collapsing building and security officers can report suspected individuals or cars.

The use of a visual iconic language, enables each of the parties involved to convey there messages to the emergency center, whether they speak the local language or not.

1.5.2 Drive-In Scenario

Another situation as at a drive-in restaurant. An establishment that enables its customers to remain in their car or other motor vehicle while being accomodated. The custom way in which the customer is being served, is by listening to the microphone attached to a menu panel on its driveway. The customer can order food and drinks. It is necessary to speak the language of the country properly. Also in this case would it be very convenient to be able to construct the requests in an easy way.

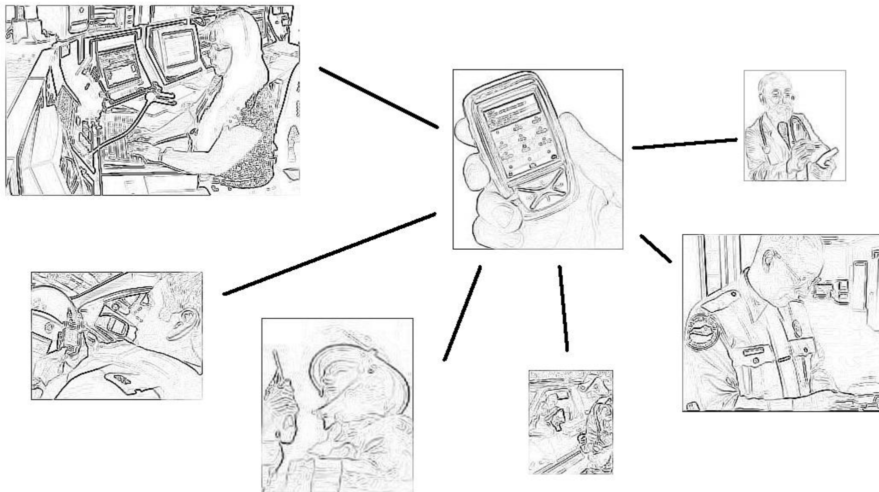


Figure 1.2: Emergency Center using Visual Messenging

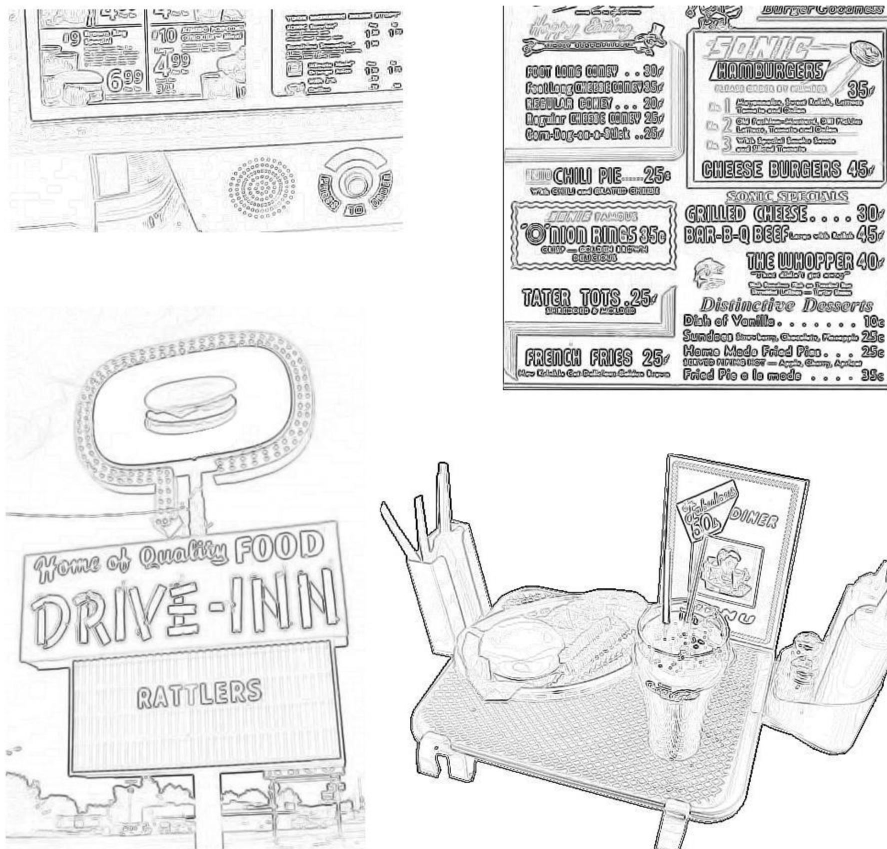


Figure 1.3: Drive-In using Visual Messenging

The customer that orders food and drinks in a drive-in restaurant reads what is available from the menu. This is a limited amount of options. It would be very convenient to be able to press these options on a kind of touch panel. Information about amounts should be added, etcetera. The situation of the drive-in restaurant does not need necessarily a full-fledged language. But it is a practical application that shows that a visual language is not merely some academic toy.

Other applications are imaginable. Like international communication between childeren, ordering plane and railway tickets at airports and railway stations (for tourists), methods of cooking and conservation on food products, weather forecasts. In many ways can the capabilities of visual languages be explored. This report does not handle such a specific domain application, but does have another goal, as previously described in subsection 1.4.

1.6 Report Structure

This report does contain three parts corresponding to phases listed in the project phases. The first part will be dedicated to a summary of the performed literature survey and a concise description of the visual languages and applications VIL and Lingua. Many issues will be omitted but can be found in the theory paper from my own hand [1] or in documents written by Leemans and Fitrianie. Topics from the literature that are used in the design process are highlighted, while judgments are postponed until later.

The second part will handle the design of the framework itself. The framework is labelled VilAug and does contain two parts, IconNet and IconMessenger. Especially the IconNet application will be the topic of this thesis. Important design decisions are discussed in this part of the report. A user case diagram will be drawn to increase understanding about the scope of this project. A functional description of the different modules in the application will be given. Also the design of the language will be underpinned by reasonable arguments inspired by the theory survey.

The third part of this report does describe the implementation of VilAug. Discussion of minor design decisions will also be postponed until this point. The report ends with an evaluation and conclusion in its last and fourth part.

1.7 Report Syntax

The layout in the report is a little bit adjusted to highlight certain specific important elements, namely:

Assumption 2. Certain assumption.

An assumption is argumented, but not proved. Improvements on the theoretical or software model may stem from thorough investigations of these assumptions as a particular research item.

Fact 3. *Certain Fact.*

A fact is a certain observation that most people would agree with and observe likewise.

Definition 4. Certain Definition.

A definition is a certain term or formulation that is used in the report as short term for the connotated description.

Recommendation 5. *Certain Recommendation.*

A recommendation is a particular interesting research topic, that does not directly solve the problem described in the problem description (see section 1.2) or deserve an own research project on their own. Practical recommendations are to be implemented in future releases of VilAug.

Remark 6. Certain Remark.

A remark is certain information that is of particular interest to particular target audience or in a specific situation;

Example 7. Certain Example.

An example is a certain description of an exemplary situation that is used for a large part of a text.

These style elements are used throughout the entire text. In the text at times screenshots and pictures from others are used. Credits can be found in appendix D on page 163.

The next part will start with the theory as described in section 1.6.

VilAug
IconMessenger IconNet

VilAug IconMessenger IconNet

Part I

Literature & Visual Language Study

Summary Part I - Literature & Visual Language Study This part of the report contains one chapter about the literature, chapter 2. It describes the theoretical background of visual linguistics. This entails cognitive psychology, general linguistics, gestural linguistics, semiotics and ontology engineering. This came to the surface in an extensive literature survey preceding this thesis [1]. The chapter contains three sections, each section containing material more relevant to the topic of this thesis. Section 2.1 starts with background theory. Section 2.2 describes theory that is explicitely used in following parts of the report. Section 2.3 contains an overview of current visual languages.

This literature in this part of the report will be followed by part II that describes the theoretical model and software model. Especially that part will contain many references to this part about the literature.

VilAug
IconMessenger IconNet

VilAug
IconMessenger IconNet

Chapter 2

Literature Survey

Is a verbal language and a sign language processed in a same way? How fast is it possible to communicate? What is a real language? What kind of grammar should a visual language have? How many symbols per minute can language users handle? How can a sound or gesture imitate that what it means? How does something get meaning? What is a conceptual model? What is an ontology? What are the properties of an ontology?

Section 2.1 with background information about *cognitive psychology* tells about working memory and brain cartography. It also contains *general linguistics* with topics like double articulation, universal grammar and Chinese script. Section 2.2 contains two most relevant disciplines. *Sign languages* will be handled in detail: orthographies (writing systems), its iconicity, amount of symbols, its modality (sensorial dimension) is brachial-visual instead of oral-auditorial. All these characteristics have parallels with a visual iconic language, and are therefore extremely interesting. The other discipline studied into detail is the area of *ontology engineering*, enriched with a *semiotic* (study of signs) and philosophical flavour.

Section 2.3 will describe the visual languages that currently exist. An overview shows that actually already more than a dozen of these languages saw the light. Two languages, VIL and Lingua, used later on as proofs of concept in the demonstrator - in the ontological and software model - will get particular attention.

2.1 Background Theory

The background theory will be limited to relevant topics and terminology in the disciplines of Cognitive Psychology (subsection 2.1.1) and general Linguistics (subsection 2.1.2). It will not return so prominently in the created model and framework as the literature topics in subsection 2.2 that contains applied theory.

2.1.1 Cognitive Psychology

Which spatiotemporal maps of brain activity exists in regard to the linguistic domain? What kinds of models exist in cartographing the brain areas in regard to the linguistic domain? What kind of brain models exists that handle lin-

guistics? Answers to these questions will use terminology and reveal facts that are needed to locate a visual iconic language in this domain; and to formulate questions, restrictions and expectations. The working memory of sign languages contains four chunks of information (see subsection 2.2.1). To understand such statements an introduction to these disciplines is necessary.

The cognitive psychology does have models that try to decompose the brain in areas where certain linguistic tasks or subtasks are performed. The Wernicke-Lichtheim-Geschwind model is well-known. A composition of the working memory is given.

2.1.1.1 Spatiotemporal Linguistic Models

One of the most famous spatiotemporal models of the brain is the functional Wernicke-Lichtheim-Geschwind model that became popular in the 1960s but dates back for almost 150 years. It is a lesion-based model, what means that it is created by observing certain language deficits. It assigns to two areas in the brain certain linguistic capabilities (and an intermediate channel). An anterior[1] area is called the Broca's area and a posterior[2] region - behind the ear - the Wernicke's area. This model assigned *motor images* for speech to Broca's area and *acoustic images* for words to Wernicke's area. Damage to the former causes a deficit in speech *production*, while comprehesion remains intact. Damage to the latter creates *comprehension* problems, while speech remains fluent. This model is outdated because it faced many problems. Some aphasia types where not accomodated, it was rather underspecified and contained severe anatomical mistakes. You can read more about this from Poeppel and Hickok [4].

Except for lesion-based modelling, there are also other ways to investigate the brain. There are also models created that are based upon scanning the brain of intact individuals that are performing tasks. Techniques that are used are fMRI (functional MRI) and PET scanning. A third type of model is the eletrical stimulation model, in which certain brain areas are stimulated and the corresponding behaviour is observed. From all these studies certain *general characteristics* of the linguistic system have been found. These characterists are uncontroversial and commonly accepted.

Take for example the neurophysical "multicomponent" model of *working memory* of Baddeley and Hitch. It contains three components: the *phonological loop* with short-term memory and a rehearsal process (so that we can hear a whole sentence, without forgetting the first words), the *visuospatial sketchpad* where spatial and visual information is processed in parallel and the *central executive* to coordinate activities of the working memory. The phonological loop was supported by the discovery of several effects:

- Phonological suppression effect: when a person has to remember (visual) items while repeating simple words like "the", or "one", "two", "three", the retention of the items is surpressed;

- Word length effect: the amount of chunks the working memory can contain[3] depends of the length of the words; "word" is easier to remember

[1] actually "inferior frontal" area, the online anatomic maps of the java 'Anatomy Browser' of the Harvard website helps to find the neurophysical locations in medical jargon

[2] "superior temporal" region

[3] working memory span

VilAug IconMessenger IconNet

than "linguistics" (it is supporting because in a certain time short words can be rehearsed more often than long words);

- Phonological similarity effect: there is greater memory interference for (visual) items that sound similar (people use thus a phonological code to remember strings of letters).

Until here the discussed spatiotemporal models capture relations between brain areas and linguistic functions. More on this topic can be read from Haberlandt [5]. Another topic of study is the way the brain handles linguistic information itself. How does a brain encapsulate grammatical rules in brain structures? How does the brain connect two concepts into one sentence? These issues are topic of the discussion in the hot debate between connectionists and symbolists. A short overview will be given in subsection on this page.

2.1.1.2 Cognitive Models

There are two mainstream research strategies in the field of cognitive science. The *symbolic paradigm* regards perception and the reasoning behind it as manipulation of symbols. The symbols could refer to exterior objects and therefore carry semantics. The symbols were taken by cognitive systems and manipulated and transformed according rules. The *connectionist paradigm* regards cognition as an interplay between elementary units in a cognitive system, that influence each other in a local way. A dynamic system arises and when it is stabilized performs it cognitive tasks. Perceptrons were the first systems that were built with this idea in mind.

It is important to realize that these two approaches exists. Statistical methods, self-organization, neural networks at one side of the coin, functions with arguments, grammatical rules, ontological networks at the other side of the coin. Dyer [6] who created an intermediate system that combined both approaches. According to him has the distributed connectionist paradigm the following advantages:

- Automatic learning and generalization: the system modifies its behaviour by iteratively comparing input with desired output; the programmar does not need to know the actual algorithm;
- Associative memory, fault tolerance: only a few clues are sufficient to retrieve a complete memory pattern, the network itself may also be damaged without abrupt malfunction, it degrades smoothly;
- Smooth parallel constraint satisfaction: instead of a chain of 'hard' rules, a lot of soft requirements are more or less satisfied in parallel, like stochastically settling in a minimal energy state;
- Neural plausibility: the brain inspired the idea of neural networks;
- Rconstructive memory: memories are stored in the same place, repetition of old data is needed to maintain old memories, but less repetition is needed during relearning; forgetting is due to interference effects.

The same author [6] lists also attractive features of symbol processing systems, namely:

VilAug IconMessenger IconNet

- Tokens and types: instances and types can be kept separated in symbolic systems (while cross-talk occurs in connectionist systems);
- Inheritance: from learning rules about types, facts about instances can be deduced (if humans are mortal, and Aristotle is a human, follows that Aristotle is mortal);
- Virtual reference: a symbol structure can point to another structure in a distant part of the memory (the same person - stored in a certain place - can be used twice in a sentence with dynamic binding; it's like pointers in informatics);
- Structure and composability: with pointers recursive structures can be built; even on the fly (like grammatical rules; and rules can be kept separated from the facts too);
- Variables and structures sensitive operations: with pointers variables can be propagated from one structure to another;
- Communication and control: with pointers output of functions can be used as input of other functions;
- Memory management: reusability of memory structures is possible (while the amount of hidden elements is - often - fixed in connectionist systems and imposes limitations upon the ease of adding new memories to old ones).

Some of the points seem to be other ways to state the benefits of one characteristic, namely having a pointer system. But it is clear that both systems have their pros and cons. The system needed to solve a particular problem depends on the problem type. See subsection 2.2.2 about ontologies and in general part II to read about which paradigm is used.

2.1.2 Linguistics

Linguistics is a very broad area and it's impossible to cover it but superficially. In this subsection certain linguistic traits will be discussed, like double articulation and grammar. Also terms like iconicity, phonemes and orthography will be defined. And some space will be dedicated to Chinese script in particular.

2.1.2.1 Language Characteristics

What is exactly a language like normally spoken? It should be distinguished from languages that make use of gestures, music, nucleotides or images. Although, languages that uses phonetics and words, but also can appear visually by means of a script.

Definition 8. A *verbal language* is a language that is meant to be spoken or written in a certain script and heard by its users. It is commonly in the oral-auditorial domain, but can also be written with symbols.

There are at least three phenomena that are characteristic for a language and elevates it from a mere (notation) system. Here will not be argued that these characteristics are prerequisites for calling something a language. The linguist Everett studied for example the Pirahã language in the Amazone for many years and questions even the existence of recursivity in human language (see Anna Parker [7]). The following three - generally considered - linguistic traits will be studied:

- Double articulation: first articulation with morphemes, second articulation with phonemes;
- Recursive grammar: restrictions at the morphological level, containing for example relative clauses;
- Orthography: notation system.

A language does have double articulation (phonological and morphological). A term stemming from Martinet's structural linguistics (and also coined by semioticians). At the level of first articulation contains the system the smallest available *meaningful units*, labelled *morphemes.* It is this level where grammar plays a role. And grammar opens the gates to recursivity. One of the forms of recursivity in English are relative clauses formed with "that". For example in the sentence: "the dog that chased the cat that jumped into the tree that fell down".

At the level of second articulation the system contains *minimal functional units* that have no meaning in themselves. These minimal pairs, are called *phonemes*. They can be discovered by analyzing contrasting features. The Japanese do not distinguish the "r" from the "l" in their language, so these letters are phonetically the same in Japanese. An *orthography* is simply a notation system. English does have (like many western languages) an alphabetic script.

Chinese has a more interesting orthography from the viewpoint of a visual linguist. But before we observe some characteristics of Chinese, an important term is asking for a definition. Namely, the recurrent term that even the title of this report carries: *iconicity.*

2.1.2.2 Iconicity

Some additional facets of iconicity will be handled in the subsection about semiotics, but a layman definition of iconicity will be sufficient over here.

Definition 9. Iconicity is resemblance of a thing with the utterance used for it. Most familiar are onomatopoeia, refering to something by imitating the sound, like "bang" or "moo" or in Japanese "dokidoki" for heartbeat.

Neil Cohn [8] visualized a gradual scale from iconic to sound-based utterances. Figure 2.1 defines a dimension from iconic to sound-based. A figure with more context is given in appendix A on page 155. It is not from iconic to non-iconic, because there are several forms in which a sign can be non-iconic. The abstraction "mandate" at the right is not purely arbitrary. It uses facts of pronounciation in a certain language and combines them like in a *rebus*. The symbol for "man" is combined with the symbol for "date" and becomes an entire new symbol.

VilAug IconMessenger IconNet

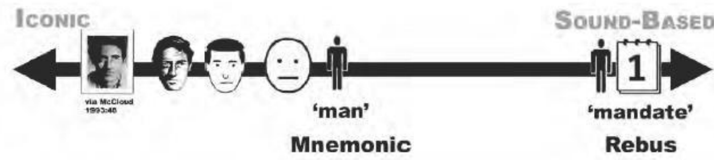


Figure 2.1: A gradation from Iconic to Sound-Based symbols

Sign language does have another way in which it is iconic, more about that in subsection 2.2.1. An interesting question stems from linguists like De Saussure who regard arbitrariness as the most characteristic property of language: "Should a language not be sound-based instead of iconic?"

An interesting study in that regard is the study of Gasser [9]. He used neural networks to study the effect of the association between form and meaning. Iconicity can be seen as a systematic relationship, as correlation, between form and meaning. In an iconic language *less has to be learned*. The form-meaning pairs are generalized towards a function with only a few parameters. Only these parameters have to be learned. In layman terms: iconicity is about generalizing, not about intuition.

In American Sign Language a *fast* movement stands for *VERY*_SLOW as Wilcox describes [10]. The intensity (VERY) is iconically represented by the type of movement (fast). Gasser found that the more forms has to be mapped to meaning the more beneficial an arbitrary mapping became.

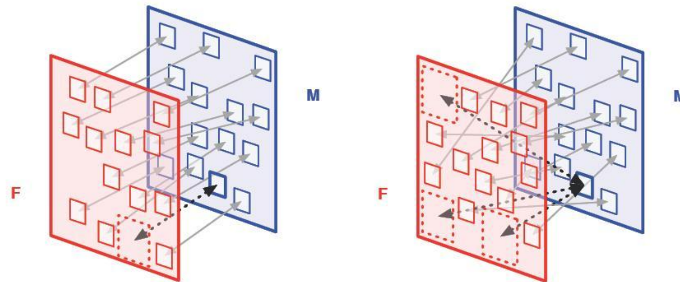


Figure 2.2: Iconicity in form-meaning mappings

The effect of an association between form and meaning was moreover also influenced by the amount of used dimensions. The more dimensions, the less beneficial an arbitrary mapping became. This study gives good reasons to assume the following:

Assumption 10. Iconicity is not necessarily detrimental to the linguistic nature of a communication system. And iconicity can be used in a language if it has enough perceptual dimensions.

This is no guarantee that an iconic representation of each imagineable concept can be created. It does however indicate that the nature of the communication channel can influence the *amount of iconicity* in a language. A verbal languages, does not contain only speech. It also has a visual counterpart: *script*..

VilAug IconMessenger IconNet

Iconicity is especially salient in Chinese script. A few facets of Chinese script will be discussed in the next subsection.

2.1.2.3 Chinese

The rebus in figure 2.1 is a good introduction to the Chinese script. Words in Chinese script are composed of two characters, that each have some meaning on their own. Most often one of this characters conveys semantics, while the other conveys a phonological clue. Chinese is therefore not semasiographic, what means that it does not relate tokens to meaning without making use of a specific spoken language. The value of its phonological facet is however dubious when it is used by Japanese, that do not know how to pronounce them. The pictographic state of many symbols is studied in detail by DeFrancis [11] who composed an overview of the type of characters in table 2.1.

Table 2.1: DeFrancis' overview of Chinese Characters (only percentages are shown)

Principle	Shang Dynasty	2nd Century	18th Century
pictographic	23%	4%	
simple indicative	2%	1%	3%
compound indicative	41%	13%	
semantic-phonetic	34%	82%	97%

The two types of "indicative" principles, are used to describe a kind of resemblance, also sometimes called *diagrammatic iconicity*, where the characters are not merely pictures of the objects they refer too. This is often the case for abstract concepts. The numbers one to three are for example presented by one to three horizontal lines. The "compound indicatives" use multiple symbols to receive the notion of a generalized, abstract concept. The symbols for "sun" and "moon" form the concept "bright". The current characters are almost all "semantic-phonetic"; they use the discussed rebus principle.

Summarized, Chinese script is not a visual iconic language with symbols like naively can be thought. Over the years many symbols incorporated sound elements. Why is this the case? Is there a correlation between the phonetic value of a symbol and its amount of strokes? Is there a correlation between the amount of strokes and the abstractness of symbols? Is there a trade-off between the amount of symbols we can remember and how abstract they are? Can phonetic symbols be reused more often? The amount of characters in Chinese is around 50,000, while the average educated Chinese knows only about 6,000 of them. DeFrancis argues that this is possible because of the rebus technique. But is this really the case? Hence, the following recommendation:

Recommendation 11. *Investigate why the Chinese script incorporated more phonetic elements over the years.*

This can be related with the printing press, where a limited amount of symbols reduces the work load. The exact reason of the shift to phonetic-based symbols in Chinese (but of course also in other languages) is unknown. It is presumptuous to derive conclusions in regard to the influence of the contemporary electronic keyboard - and other computerized technologies - in regard to

VilAug IconMessenger IconNet

symbol manipulation. A visual iconic language *might* profit from the existence of fast access methods to large sets of symbols, that became available with the information revolution.

2.1.2.4 Chomskian Grammar Types

Language can be ordered in certain hierarchies in regard to their rewriting rules (grammatical rules). Chomsky designed such a hierarchy of languages with different grammatical abilities. The Chomskian ordering scheme places *regular languages* at the bottom of this hierarchy. Regular languages can be recognized and parsed by a finite state machine [FSM] that has no memory. This kind of language has no *left-recursive rules*. Left-recursive rules have the form of equation 2.1.

$$A \rightarrow A\alpha|\beta \tag{2.1}$$

The term at the left is the first term of the production rule at the right. The difficulty of these kind of grammars lays in the fact that a machine that has to parse such sentence has to remember phrases. A *top-down* parser is like an FSM and does not recognize left-recursive rules. There are several solutions: An option is to add a rewriting module that generates right-recursive rules from left-recursive ones like described by Moore [12] originally from and named after Paull. Another way is to add memoization (a kind of caching) of top-down parsing that makes the parser similar to a chart parser as described by Johnson [13].

It is also possible to use a chart parser itself. This parser switches continuously from top-down to bottom-up parsing. It knows how to handle *context free grammars*, a grammar that does have left-recursion as well as right-recursion. These grammars are higher (type 2) in the Chomskian ordering scheme. This kind of grammar can be parsed with a push down automaton [PDA]. The PDA is an adapted version of the FSM. It does have an additional memory bank or stack. Switching from state is controlled by input parameters and the previous state, but also by the item at the top of the stack.

There are even more advanced types of grammar but fortunately these kind of grammars won't play a role in our visual languages. In designing the grammar someone should be aware of the types of grammars that exist. The complexity of the grammar will impose requirements upon the software model (see subsection 3.3.3 on page 68 and section 4.10 on page 98).

2.2 Applied Theory

The background information in section 2.1 provided us with needed terminology. To be able to understand the position of a visual iconic language, subsection 2.2.1 dicusses sign languages. The model and framework arose from ontology engineering and philosophical and semiotical insights. These disciplines are described subsequently in subsection 2.2.2.

2.2.1 Sign Languages

Gestural linguistics is the discipline studying sign languages, the language of the Deaf community. Signing is a well known way of multimodal communication. Colours, cords, whistling and lip reading are all used for cummunication, but the theoretical framework regarding sign languages is the most extensive. There are many sign languages. Some of them are the American Sign Language [ASL], the Australian Sign Language [Auslan], the Dutch Sign Language [NGT]. The user of a certain language can not understand the user of another language without learning the other language just like in the case of verbal languages. The British and the American signers do not understand each other. A sign language contains 3,000 until 10,000 signs. Working with such a large set of symbols asks innovativity. It is possible to ask questions about the richness of sign language, its conceptual accuracy, its vocabulary size, eloquence, etcetera. And sign language is used in the disciplines of chemistry, mathematics and other sciences for example. But, let us regard the set of characteristics that we considered to be quite decisive, although not prerequisite for a language, like described in 2.1.2.1 and 2.1.2.2.

Does sign language exhibit the characteristics discussed earlier? It does seem so. Sign language has indeed double articulation. The morphemes in sign languages are classifiers, agreements, etcetera. Also has sign language relative clauses equivalent to “that”. And at the level of second articulation has sign language minimal pairs, chiremes (rather than phonemes) like handshape, orientation, movement, etcetera. Sign language does also have its orthographies, like Stokoe and SignWriting. Such orthographies enhance its standardization. SignWriting uses a certain lexicographically ordering of its signs. An example of SignWriting is given in figure 2.3.

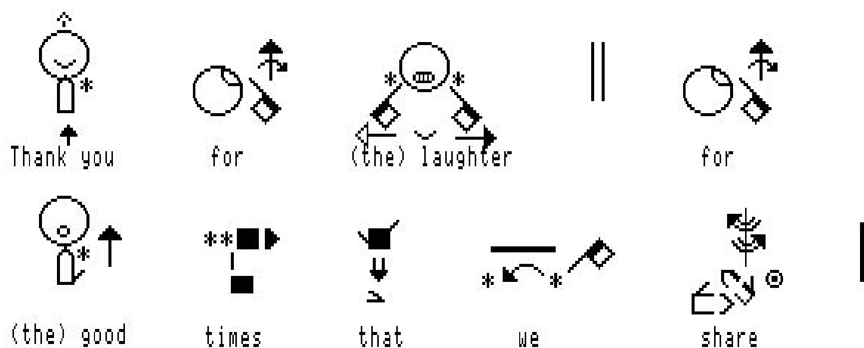


Figure 2.3: SignWriting Example about Friendship

One of the languages that have a higher grade of iconicity is a sign language. However, gestures are not entirely iconic, nor is the concept iconicity uniquely defined. Dynamic iconicity plays a role in narratives, frozen iconicity in poems (sign language does have its poets!), and no iconicity in conferences. Sign language does already have the mentioned five characteristics.

Although sign language is communicated in a lower rate, the actual conceptual transmission rate is equal to a spoken language. Breakdown by replaying

VilAug IconMessenger IconNet

users on a higher speed is (equal to a spoken language) at 2.5 until 3 times normal communication speed.

Auditory language areas are counter-intuitively also used for signing. This is because grammatical processing is needed for both. There is phonological loop rehearsing input information in sign language as well. The working memory supports therefore both speech and sign skill. The reading (visual) and hearing (auditorial) capabilities of people suffering Wernickes aphasia ar different. So visual and auditorial language processing is different in another way. There are persons with an aphasia that experience difficulties with Japanese kana characters. Kana relies much more upon phonetics than kanji that relies upon semantics too.

2.2.2 Ontology Engineering with Semiotics & Philosophy

We start our quest in the field of ontology engineering by delving first in the study of signs. This discipline is called semiotics. There were two major players in history. The linguist De Saussure and the philosopher Peirce (pronounced purse) both lived in the second half of the 19e century. Another famous representant of this discipline is Umberto Eco, the writer. Semiotics has much to do with semantics, but is less prone to look at communication from a linguistic angle. The study investigates not merely the meanings in isolation, but how meanings arise in their contexts. What makes something meaningful? Can concepts be described by other means than by words? What is the relationship between a concept and that where it stands for?

2.2.2.1 Terminology

One of the main entities in an ontology is a concept:

Definition 12. A *concept* is an entity with an identifying label and certain content. This content can be represented in different ways. Concepts are the same across languages, but can be translated in multiple ways, and represented differently. A concept is related to other concepts.

An ontology can be represented as a relational database, a relational table or a graph.

Definition 13. A *graph representation of an ontology* is a graph construct that contains vertices and edges. The concepts in the ontology map unto the vertices of the graph, the relations between the concepts in the ontology are modelled by the edges.

This definition is too coarse to be maintained throughout the report, later on in this section will a more sophisticated definition be proposed.

2.2.2.2 Requirements

According to the W3Consortium [14] several requirements exists that an ontology should obey. To these points belong the following (with a little bit modified formulations):

1. Two concepts (in an ontology) must be distinguishable by their unique identifiers;

2. Ontologies must be extendable by other ontologies and reuse their concepts in a transitive way (if B incorporates C, than involves extending A by B the incorporation of C by A);

3. Versioning information about the ontology should be stored; backwards compability but also deprecation of certain concepts / identifiers;

4. Classes have to be defined by (at least mechanisms as advanced as) subclassing and boolean operations;

5. Properties have to be defined by (at least mechanisms as advanced as) subproperties, domain and range constraints, transivity and inverse properties;

6. Classes should also function as instances. If Orangutan is an instance of Specie, it should still be able to have several individual animals as instances itself. The perception of the user defines if a certain concept is a class or an individual / instance;

7. An XML serialization syntax.

These requirements are more or less web-oriented, but many of them can just as such be used as favourable characteristics for ontologies in general.

2.2.2.3 Biases

One of the first issues a semiotician like Chandler [15] points out is the phenomenon of logocentrism or verbocentrism. This interpretive bias privileges verbal linguistic communication by eye (written words) or by ear (heared words) over non-verbal forms of communication. Another interpretive bias is phonocentrism. Speech is given in some way a higher status than writing. In literate societies there is the tendency to lean towards graphocentrism. Written language is seen as a standard and oral language is subordinate. Like Ong comments (quoted by Chandler):

> "Because we have by today so deeply interiorized writing, made it so much a part of ourselves... we find it difficult to consider writing to be a technology. [...] Freeing ourselves of chirographic and typographic bias... is probably more difficult than any of us can imagine."

Recapitulating, there are three prejudices in linguistics. They can be reformulated in a cautious way in the form of assumptions. There might exist reasons to be verbocentric, phonocetric or graphocentric, but it is assumed in this report that this will not be the case:

Assumption 14. There is no a priori reason to prefer verbal over non-verbal communication. And verbocentrism should be guarded against.

Verbocentrism can come in secret ways. Grammatical categories of verbal languages can for example be used as a template for sign languages. But, actually this use should be substantiated. A similar assumption based also upon the fact that written and sign languages exists.

VilAug IconMessenger IconNet

Assumption 15. There is no a priori reason to prefer oral over visual communication. And phonocentrism should be guarded against.

And the last assumption is the other side of the coin:

Assumption 16. There is no a priori reason to prefer visual over oral communication. And graphocentrism should be guarded against.

There are reasons why in certain situations some kind of communication can be preferred over another. In the context of physical or mental disabilities, or when certain properties of a domain are investigated in detail. But a priori no system can be preferred over another. In this thesis this will be the (unargued) assumption.

2.2.2.4 Semiotics

To understand the nature of a concept and to understand the idea of relational meaning we have to explore the basics of semiotics. Semiotics is a fashion word for the study of signs. There are two dominant models of what consitutes a sign. Each one developed by a person that became one of the foundingfathers of semiotics itself. Namely, De Saussure who suggested a dyadic form, and Peirce (pronounced purse) who suggested a triadic form. Let us start with De Saussurean definition of the sign:

Definition 17. The *De Saussurean sign* is dyadic and contains a signifier and a signified. The former is the form of the sign, the latter the concept it represents.

Let us explain this with an example: the white flag on the battlefield points at a situation of peace. Like can be seen in figure 2.4. The white flag is the signifier, the situation of peace the signified. If the (concept of a) flag or (the concept of) peace did not exist, the sign would not have existed either.

Figure 2.4: The white flag and the situation of peace

Another important Saussurean observation is that the concept of meaning is *relational* rather than referential. There are entire *systems* of signs. Each sign on itself does not make any sense. Only in the context of the other signsn do they make sense. The word “sheep” for example is known because a large amount of the English sign system is known to the user. Its Saussurean value however can be different from the French “mouton” that can also be used for the meat of a sheep: “mutton”.

VilAug IconMessenger IconNet

Saussurean semioticans emphasis the *arbitrary* relationship between signifier and signified. Saussure considered it as the first principle of language. No two languages categorize reality in the same way. This does not mean - of course - that signifying systems are socially or historically arbitrary. The relationship is conventional. According to Culler those following the Saussurean model have tended to avoid the familiar mistake of assuming that signs which appear natural to those who use them have an intrinsic meaning and require no explanation. We will continue the material regarding arbitrariness and its antithesis iconicity later on. Let us first continue with the Peircean sign:

Definition 18. The *Peircean sign* is triadic. It has three constituents: the representamen, the interpretant and the object (or referent). The representamen is the form that the sign takes, the interpretant is the sense being made of the sign, and the object is that what is referred to.

Let us use the same example again. The representamen is the white flag (the form of the sign), the referent is the situation of peace (what the sign stands for) and the interpretant is the idea that the white flag stands for peace (the sense being made) By the way, the interpretant is the mental structure in the brain of the interpreter rather than the interpreter itself. The existence of this third component, the interpretant (the interpretation that the representamen refers at the referent), can be a matter of life and death on the battlefield.

This tiny difference has several consequences. Firstly, the relationship between representamen and referent is made explicit. It is possible that there are two different signs that only differ in their interpretant. A good example gives Frege, who points out that the same object "Venus" can be seen as the "evening star" and as the "morning star". Secondly, the representation of De Saussure's sign lends itself more to see a difference in the ontological reality between the signified and the signifier. While, the referent can exist in the objective reality as well as the representamen. The third point is that it is easy to see the interpretant as the representamen of a new sign again. For example when this memory structure (commonly refered to as thought) leads to another thought. This matches a bit with the discussed Saussureans notion of the relational nature of meaning. Words in a dictionary are explained by other words, that are explained by other words, ad infinitum. Fourthly, the Peircean sign is less anthropocentric. It fits the domain of biosemiotics where interpretation by non-human systems occurs.

Peirce made many other distinctions like different types of relationships between sign vehicles (representamen) and their referents. I will list them overhere, because the notion of iconicity stems originally from here:

- Symbolic: the relationship between signified and signifier is conventional (for example flags);
- Iconic: the signifier resembles the signified in some way or aspect (for example metaphors);
- Indexical: the relationship between signified and signifier is causal (for example footprints or pain).

According to Peirce is a sign an *icon* insofar as it is like that thing and used as a sign of it. Unlike the index, the icon has no dynamical connection with the object

VilAug
IconMessenger IconNet

it represents. Chandler tells that Langer argues that even a picture is essentially a symbol, and not a duplicate, of what it represents. Pictures resemble what they represent only in some aspects, other aspects are conventional (like the use of light, camera angle, etcetera). Much more about semiotics can be read in Chandler's excellent introduction [16].

Now there is a basis about what a concept is, and how relationships in and between concepts can be seen, it is time to see how the concepts can be collected into datastructures. A framework that has to store several visual languages, contains such a datastructure, in the form of an ontology. That is the topic of the next subsection.

2.2.3 Current Ontologies

This subsection will place ontologies in a conceptual spectrum first. Secondly, a specific kind of ontology, Sowa's conceptual graphs are highlighted. After that other kinds of ontology types and typical characteristics are briefly noted. It ends with examples of current ontologies and extensions upon them. This subsection is not meant to give an entire bird-view of ontology engineering, but emphasizes particular facets that played a role in the design process. The implemented ontology can in this way also better be placed when something of the spectrum is shown. And last but not least, some of the mentioned theory functioned especially as inspiration because of its generality or originality.

2.2.3.1 Beyond Ontologies, the Entire Description Spectrum

Ontologies are a kind of data representation. Where to embed the ontology in the entire field of human knowledge is given by its level of description. This distinction comes from Guarino as cited by Farrar and Bateman [17], see table 2.2.

Table 2.2: Levels of description suggested by Guarino

Level	Primitives	Interpretation	Main feature
Logical	Predicates, functions	Arbitrary	Formalization
Epistemological	Structuring functions	Arbitrary	Structure
Ontological	Ontological relations	Constrained	Meaning
Conceptual	Conceptual relations	Subjective	Conceptualization
Linguistic	Linguistic terms	Subjective	Language dependency

Like can be seen, there is a difference between an ontology and a conceptualization. The kind of ontology that will be developed will lean towards the conceptualization part of the spectrum. This means that the interpretation becomes less constrained and more subjected and therefore more customizable. Less us first however highlight some specific ontology, that seems to be based most on the ideas of one of the already mentioned semioticians.

2.2.3.2 Conceptual Graphs

The semiotician Peirce developed a model of existential graphs. In this world of graphs a blank page denotes truth. Closed curves around propositions (denoted

by symbols) indicate negation or complementation. So drawing Q inside P with circles around each of them stands for the conditional $P \rightarrow Q$, when they both have circles around them and are part of a bigger circle they stand for the disjunction $P \vee Q$. This model together with semantic networks forms the basis of conceptual graphs, as described by John F. Sowa. In conceptual graphs sentences like this can be described: "A cat is on a mat." This can be notated by a short-hand script as $[Cat] \rightarrow (On) \rightarrow [Mat]$ called the linear form, or as a graphical representation:

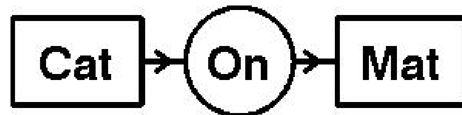


Figure 2.5: Conceptual Graph of "A cat is on a mat"

An conceptual graph can contain quantifiers, like the existential ∃and the universal ∀. So conceptual graphs contains propositions in a graphical form, it contains knowledge. Interesting is also that relations in conceptual graphs can be triadic. The between relation is indeed 3-ary connected to three arcs.

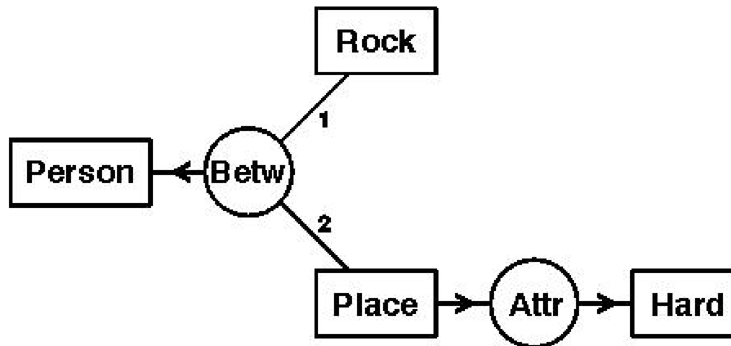


Figure 2.6: Conceptual Graph with triadic relation

Relations are in conceptual graphs regarded as first-class citizens, and that will be an important characteristic of the forthcoming implemented ontology.

2.2.3.3 Ontology Spectrum & Characteristics

In the literature there is a difference between lightweight ontologies and heavyweight ontologies. Like Corcho, Fernández-López and Gómez-Pérez describe [18]:

> "On the one hand, lightweight ontologies [taxonomies] include concepts, concept taxonomies, relationships between concepts, and properties that describe concepts. On the other hand, heavyweight ontologies add axioms and constraints to lightweight ontologies."

It is important to note that when ontology is used in this thesis, such a lightweight ontology is meant. There are many other types of ontologies, even if they appear in the form of networks. Sowa [19] lists several types, among with are:

- *Definitional* networks emphasize the subtype or IS-A relation between a concept type and its subtype. This network is also called a generalization or subsumption hierarchy. Inheritance means that the properties of the supertype are inherited by the subtype. Definitions are true by definition, and the whole hierarchy is necessarily true;
- *Assertional* networks assert propositions. Their truth is contingent (not true under all circumstances) and additional modal parameters are used to obtain absolute truths;
- *Implicational* networks use implication as the primary relation for connecting nodes. In this way causality and inference can be modelled.

The types show important dissimilarities between ontologies. And this is about the function of the ontology in general, about what should be done with an ontology. Another question is how to model an ontology. The most salient issue is in that case the type of the first-class citizens of the ontology. What are the most important elements of the ontology? Are ontologies themselves considered as first-class citizens and should it be most important to collect ontologies together? Or, are types the most important elements of the ontologies and can instances seen as attributes of types? Or are mutations to the ontology the most important objects, and should they be stored and manipulated above all?

An overview of existing ontologies is given by Noy and Hafner [20], and besides Sowa's work they review others that are famous within the field of ontology engineering:

- CYC, a general ontology for commonsense knowledge to facilitate reasoning;
- Sowa's ontology, a general ontology that synthesizes philosophical insights;
- WordNet, a comprehensive lexical ontology, a manually constructed online reference system.

Their comparison of ontology restrictions is also clarifying. They compare the ontologies according to several characteristics:

- General: domain specificity, integration, amount of concepts, formalism, implementation platform;
- Design process: implementation details, evaluation;
- Taxonomy: types of taxonomies, ontology building blocks (things, processes, relations), treatment of time, top-level division, density;
- Internal structure of concepts: properties, roles of concepts;
- Axioms: expliciteness of axioms, form of axioms;
- Inference mechanism: existence of reasoning, manner of reasoning, beyond first-order logic;
- Applications: retrieval, user interface, applications that use the ontology.

VilAug IconMessenger IconNet

An ontology should point out which characteristics it exhibits. It is not always necessary to have a user interface or a notion of time in the ontology, because that depends on the situation.

2.2.3.4 Specific Ontologies like CYC and WordNet

The ontologies that have more general scope will get some attention. To these belong CYC (abbreviation of encyclopedia) and WordNet. CYC is created by Lenat and others and contains more than 10,000 concepts. It does have an upper hierarchy level, with concepts like "Thing" and subconcepts like "RepresentedThing" and "InternalMachineThing" (that contains things to the local platform on which CYC runs). WordNet developed by among others Miller, contains concepts that are semantically interrelated. A basic lexical division in nouns, verbs and adjectives is maintained and the relationships used for each categories differ from hypernymy, metonymy and antonymy to entailment. The central object is however the *synset*, a set of synonyms. Its top-level hierarchy exists out of {thing, entity} to {living thing, organism}, {non-living thing, object}, {plant, flora}, {substance}, etcetera. In general are the top-level entities and the top-level hierarchy of each ontology - although all are meant to be general ontologies - very different!

To be aware of the limitations of an ontology, it is interesting to search for its sequences. The OMCSNet [21] is such an attempt. It enriches WordNet with concepts that are like CYC's commonsense knowledge. The authors:

> "For example, WordNet can tell us that a dog is a kind of canine which is a kind of carnivore, which is a kind of placental mammal, but it does not tell us that a dog is a kind of pet, which is something that most people would think of. Also, because it is a lexical database, WordNet only includes concepts expressable as single words. Furthermore, its ontology of relations consists of the limited set of nymic relations comprised by synonyms, is-a relations, and part-of relations."

So, there are many opportunities to improve even an ontology like WordNet used on such a large scale. The points: latticed[4] and more informal subsumptions, expressions as concepts and custom relationships are natural extensions to WordNet. The OMCSNet ontology contains concepts like "wash hair", "brush teeth" and "automatic teller machine" and is thus not anymore a *lexical* knowledge base. This is in accordance with assumption 14, that treats concepts different from mere words.

Another project is OntoWordNet [22] that takes the synsets of WordNet and tries to formal specify the conceptualizations conveyed by these synsets. Especially its ontological distinctions need refinement. The hyponym / hyperonymy relation is seen as formal subsumption[5] and sometimes as instantiation[6]. The other relationships are refined likewise. The reconfigured WordNet is apt to

[4] the structure resembles latticework

[5] the "formal" in "formal subsumption" stems probably from the term "formal concept" in formal concept analysis, that would mean that the subconcept of an IS-A relationship inherits all the "formal attributes" of its superconcept

[6] in the case of a specific country for example

VilAug IconMessenger IconNet

be incorporated in DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering).

2.3 The Visual Linguistic Field

At the end to our quest through existing theory, we land at the field of visual linguistics itself. Two visual languages will be described, namely the languages VIL (Leemans) and Lingua (Fitrianie). This because the framework VilAug can be seen as a conglomeration of the applications VIL and Lingua, and because the languages VIL and Lingua will be stored inside this framework. First, however, an overview of this field will be given and eventually some prejudices destroyed.

2.3.1 Overview

This section gives an overview of the field that is related to visual linguistics. Many visual iconic languages exist and have existed, like Bliss [23], CD-Icon [24], Isotype[25], Visual Inter Lingua[2], Vedo-Vidi[26], The Elephants Memory[**?**], Kwikpoint[27], Musli[28], MediaGlyphs[29], Sanyog[30], PIC, C-VIC and MinSpeak[31]. Some of these languages predate the computer. Others are commercialised, often targeted at disabled users. Some of them lack a theoretical basis. A few of these languages will be discussed in the next subsection.

2.3.1.1 Visual Language & Icons

The items carrying meaning in a visual language are the icons, as shown by existing visual languages. These icons can be manipulated and concatenations of icons define the meaning of a sentence. The PIC language is pictographic. The drawings look like the things they stand for and are highly iconic. The use of abstract concepts is extremely limited. The Bliss language is inspired by Chinese and is ideographic (in the sense of "compound indicative", see 2.1.2.3). It uses combinations of meaningful icons to create other icons. The MinSpeak language resembles the Bliss language but uses the rebus principle, see figure 2.7.

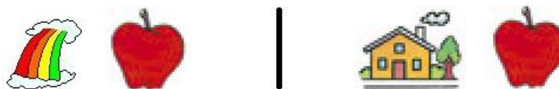


Figure 2.7: MinSpeak: (Rainbow, Apple) can mean Red and (House, Apple) can mean Grocery

This rebus principle can be taken to a higher level. The meaning carried by icons can be analyzed. Like for example in the sense of a kind of icon "algebra". Chang [32] distinguishes icon operators like:

- Combination COM(X,Y), merges icons X and Y in a conceptual way;
- Marking MAR(X,Y), emphasizes a local feature of X by means of icon Y, a conceptual restriction by local emphasis (Red in figure 2.7);

- Contextual interpretation CON(X,Y), considers X in the context of Y, a conceptual refinement by adding context (Grocery in figure 2.7);
- Enhancement ENH(X,Y), adds to X the attributes of Y (for example "low");
- Inversion INV(X), uses a specific negation icon to invert the meaning of X;
- Indexing IDX(X), extracts the most salient attribute of X (like "big") and uses its subsequently for the next icon Y.

This gives an idea how icons can be combined in different ways. And each of them doesn't have to do with phonetics at all. The meaning implied by the basic building blocks, the icons, in a visual language can be manipulated in various ways. A visual iconic language is understood in this thesis of being of such character, and not a natural visual "language" like a movie or cartoon.

2.3.1.2 Visual Language & Goal

The developers of these languages had each their own goal. Their goals are not stated as such explicitely, but the following main directives can be found:

- To facilitate disabled users in their communication, often the case in the field of AAC [Augmented and Alternative Communication] (like Sanyog, MinSpeak, Lingraphica);
- To test certain theories (like CD-Icon);
- To help children in notating dreams and thoughts (like Vedo-Vedi);
- To easify international communication (Bliss, Kwikpoint).

These goals are not very far-stretched, but one side note has to be made. A visual iconic language is not a magical way of communication. It does have its own benefits and the reason that we use our current orthographies might be based on historical grounds only. However, it is not likely that it becomes the only language on earth. It is not so, that when a language gets a large amount of practitioners, that when it reaches a certain take-off point, it suddenly will force the remaining part of the population to switch to that language too. Pool [33] demonstrated this in a computational model. The size of a (natural or artificial) language speaking group appeared to have no causal relation with its stability. Another clue is given by sign languages, that are neither universal. There are also no universal classification methods, nor universal ontologies, nor universal orthographies. So let us formulate this:

Assumption 19. A visual iconic language can have multiple goals, but universal practitioning on earth is unachieveable, hence not among its goals.

A universal language will have its illiterates too...

2.3.2 VIL

The two languages that are discussed into detail are the languages/applications Visual Inter Lingua [VIL] in this subsection and Lingua in the next subsection. They are chosen for multiple reasons:

- The languages VIL and Lingua are stored in the developed framework as proof of concept;
- The code and the user interface of the application Lingua is used for the developed application;
- The theory behind the language VIL was the inspiration to provide a solution as less verbocentric (see 14) as possible.

Let us review the language VIL.

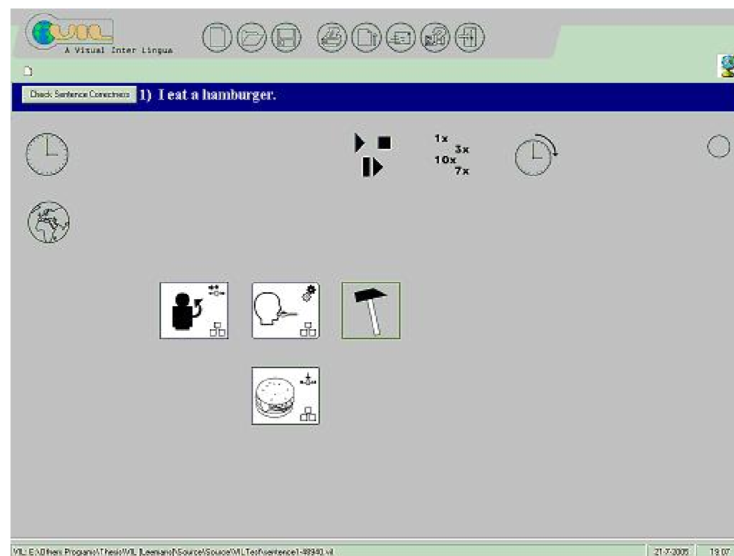


Figure 2.8: Screenshot application VIL (by Leemans)

VIL (Visual Inter Lingua) is the work of Leemans. An iconic language like VIL has a grammar and an (organized) lexicon of icons (iconicon). The grammar is inspired by (but not building on) Fillmores Case Grammar and Schanks Conceptual Dependency Theory. Case grammar uses grammatical categories that are not the famous “verb”, “noun”, “adverb”, “adjective” from verbal languages. It contains grammatical classes like “actor”, “action”, “patient”, “instrument”, “locative”, “dative” and “objective”. It does have above that modalities like “tense”, “aspect”, “form”, “mood” and “manner”. Schanks conceptual dependency theory tries to reduce a verb into more primitive constituents, labelled ACTs. For VIL the classification of verbs borrows extensively from the proposed categories by Schank. In VIL the grammatical concepts, called grammatical cases, semantic roles or theta roles, of above are taken. There is a semantic role “theme” that is like “patient”. No roles or subroles are added, like “benefactive”, “causer”, “experiencer”, “effector”, “comitative”, “measure”, “extend”, “incremental theme”, “non-incremental theme” or “holistic theme” [34].

VilAug IconMessenger IconNet

Is there still a classification of verbs in VIL? The answer sounds unexpectedly: yes! The grammatical classes like “actor” and “action” can exist independently from the notions of “verb” and “noun”. Still chose Leemans to create tables with “noun icons”, “verb icons” and “adjective icons”. The grammatical “actor” category contains subsequently the icons from the “noun” table. The “patient” category contains exactly the same set of “noun icons”, just as the “instrument” category. The “action” category contains all “verbs”, etcetera. So, the iconicon of VIL does still have these three types of concept classes: “nouns”, “adjectives” and “verbs”. Each of these classes are organized in a separate hierarchy. So VIL can be presented as 3-tuple:

$$C = \{N, A, V\} \tag{2.2}$$

The entire collection contains several sets. Each capital stands for a set of icons of a particular type. Each set is ordered in a hierarchy with only a few levels (on purpose). The highest level is - of course - what the user encounters when she starts the application. The user starts with filling empty boxes on the screen (see figure 2.8). The available choices are derived from case grammar and conceptual dependency theory. The Extended Backus-Naus-Form (EBNF) notation will be used to clarify this. (For assignments ::=, for OR operations |, for one or more items * (also called Kleen-Closure) and for optional items surrounding brackets []. Instances of a category will receive quotes.) The example with “push” uses case grammar terminology, the example with “propel” uses conceptual dependency theory:

1. ”push” ::= objective [locative] agent [instrument] dative
2. “propel” ::= actor object direction [time] [location] [instrument]

These (grammatical) categories can be seen as the upper-hierarchy in VIL. Which categories become visible depend on the kind of “action”. Each “action” does have a kind of signature, called a case frame. There are for example transitive verbs and intransitive verbs. Like this sentence with a transitive verb: “The boy kicked the ball.” And this sentence with an intransitive verb: “The boy ran.”

Only transitive verbs do have (one or more) direct objects. Except for these syntactical case frames, it is also possible to create semantic case frames. Nouns can be associated to verbs as: instruments (knife -> cut), materials (wallpaper -> attach, wool -> knit), products (hole -> dig; picture -> paint), containers (box -> hold), etcetera. These common assocations can be used to predict which items will be connected to the already selected verb. This is however not implemented in VIL.

The above mentioned categories are actually embedded in an even larger construction. For example, even before anything is drawn the user is asked if the sentence is an exclamation, a normal statement or a question. The whole sentence is as follows:

3. sentence ::= modality verb [kasus] noun_phrase

Most of these terms are self-explaining. The term kasus is used to indicate case, and especially the elements corresponding with “by” and “with” that are used

VilAug IconMessenger IconNet

to start a proposition as in “with the hands”. The term modality is about tense and aspect (a table with modalities can be found in B). Last but not least, the noun phrase can further decomposed in the following way:

4. noun_phrase ::= [determiner] [adjective | noun]* noun [sentence | noun _phrase]

With these rewriting tools many sentences can be constructed. The rule of above can contain multiple nouns. A noun modifier precedes a primary noun, like in “fire truck” or a postpositional modifier a noun can succeed the primary noun, like in “a house that seems a palace”.

The nouns, verbs and adjectives are meant for daily use. It is a general lexicon and not domain specific. The top entities of each category are shown in table 2.3. A more fine-grained distinction can be found in appendix B.

Table 2.3: Categorization of Icons in VIL (columns are not related)

	nouns	verbs	adjectives
1.	physical world	mental	lower level perceptual
2.	beliefs, customs and society	alienable possession	higher level evaluative
3.	arts & entertainment	physical location & motion	comparative / relational
4.	sports	existence	
5.	communication	identificational appoint	
6.	science & technology	involuntary	
7.	transportation	voluntary	

It is an art to come up with a classification that extends to all possible concepts, while restricting the amount of items at each level. Leemans managed to create levels around the short-term memory limitation of around 7 chunks. In VIL no particular kind of relationship is used. In the noun category IS-A relationships are often used. In the adjective category, the antonymy (good is not bad) is used. These relationships are however not used consequently in an entire category.

This is a very brief introduction to the language VIL, but should be enough to get an idea about what is needed to support such a language in a framework. Let us now review the language Lingua.

2.3.3 Lingua

The visual language Lingua can be described very briefly. It does actually reflect the grammar and words from the English language. The grammatical categories that are invented are the (verbocentric) categories of “noun”, “verb”, “adjective”, “adverb”, etcetera. A few categories are added like punctuation (question-sign and exclamatory-sign) and negation. Let us again start with the grammar and finish with the ontology or classification scheme. But to get a feeling for this application, take a look at figure2.9 first.

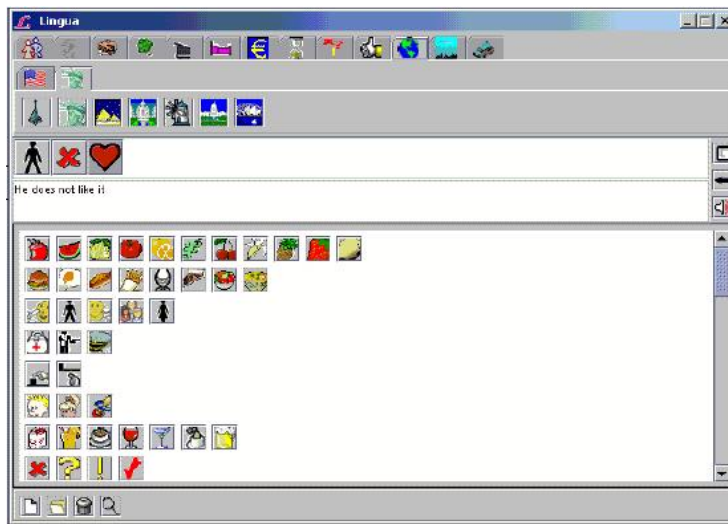


Figure 2.9: Screenshot of Lingua

In the language Lingua the types of sentences are restricted. It seems like the English language functioned as its inspiration and a kind of basic English in icon form results:

5. sentence ::= regular sentence | adjective sentence | question sentence | negative sentence | exclamatory sentence

Three examples:

6. sentence ::= "table" "two"

7. sentence ::= "direction" "museum" "map" "?"

8. sentence ::= "not" "to go" "direction"

The icons in the menu are disabled according to their grammatical class. If a certain sequence can not be followed by a "verb" every icon that is a "verb" will be grayed out. Malformed sentences can not be constructed in that way. The grammatical rules of Lingua are written in a XML file. They seem to be applied correctly, except for the case of the very first icon. Starting a sentence is possible with an arbitrary icon.

The classification is contrary to VIL, aimed at the particular situation of a toerist in a foreign country. It is based upon situations in which consulted participants indicated that speaking the native language would be appropriate. The classification consists out of 13 classes, see figure 2.4. A more fine-grained version can (again) be found in appendix B.

With this we come to the end of our theoretical research. Much have been omitted, but much has yet to come. The following part of the report will discuss the model, that uses additional terminology that doesn't make sense when discussed on its own. To explain a facet of the model forthcoming I refer for example to the discipline of graph rewriting. This is senseful for readers that

Table 2.4: Categorization of Icons in Lingua (columns have no specific meaning)

nr.	first part	second part
1&7.	people	time
2&8.	verbs	extras
3&9.	in the restaurant	adjectives
4&10	in the market	the world
5&11	in the shop	in the city
6&12	in the hotel	transportation
7	money	

know that field, but it would take to far to treat that specific topic in this part. Also certain design technologies or patterns will not be appropriate overhere, but postponed until later. They will be interesting to the programmer only. In the model much new material will be provided. Not everything is entirely new, but it would take too much time to do more research into existing literature. Enjoy reading the next part, it sheds new light upon especially the field of ontology engineering.

Part II

Model & Architecture

Summary Part II - Model & Architecture This part of the report will describe the two types of model that have been designed. Chapter 3 contains the theoretical, also called ontological, model. Design decisions in regard to the nature of iconic concepts, or the manners of parsing grammar are described. At times the theory from the previous chapter will be referenced for further background information. At some times additional theory will be used to motivate decisions.

Chapter 4 is about the software model. The term "architecture" is also used to distinguish the theoretical or *ontological model* from the *software model*. The chapter that describes the architecture is about application and function simplicity, reusability, modularity and speed. Other decisions have to do with data standards. It contains the model in the form of a UML description, it describes what type of grammar parser is used, it describes browsing strategies for the ontology.

The actual description of parts of the application and its syntax is given in part III about the *implementation*. That part is meant for developers. It contains language dependent information, a GraphML API (Application Programming Interface), and sourcecode. If this part of the report does not contain enough detail, it can probably be found in the next part.

Chapter 3

Ontological Model

This chapter describes the developed model. It contains all information that can be read independently from any notion how this would result in useful applications. Although, one has to keep in mind one point. And that is that relations between icons are is to know which subitems will follow their superitems in an icon menu like “Save” follows “File” in an application menu. The chapter starts in subsection 3.1 with some general definitions and ends with the basic building blocks of a visual language: an iconicon, an ontology and a grammar. The ontology and the kind of elements it contains - concepts, relations and grammatical classes - are reviewed upon their characteristics. This subsection discusses the pros and cons of the model as a whole and its graph representation in particular. It ends with a description of how grammar is handled in this model.

The second part of this chapter (subsection 3.2) makes a translation from the abstract material about the ontology, the characteristics of the used relations, the possibilities of a graph representation, to actual examples of how icons can be subclassed, how an icon can be assigned to a grammatical class, how an icon can be hidden in the menu, and what kind of mutations can be applied to the graph.

And (again), details about the software model (the architecture) will be postponed till the next chapter.

3.1 Model Description

This subsection contains also theory, but this part relies much less upon existing theory from the field and the model is self-made for a large part. The conceptual graph theory [19] inspired me, as well as other theories. But most of the theory discussed in this chapter is provided to give the developed model its place in the literature. Important terms and thoughts are captured in (numbered) definitions, facts, assumptions, etcetera. Reading through them should also give insight in the purposes of this model.

3.1.1 Formal Definitions

A visual language does not only exist out of icons: the graphical elements. It does entail much more. Firstly some terms will be defined.

Definition 20. An iconography is a package of images on the disk, currently in .gif or .jpg format. Different iconographies represent different icon styles. An iconography can be designed by a graphical designer.

An iconography can be compared with a *font* for a verbal language. A font does also have the freedom to represent a letter like this X or like this Ξ or to use numbers in arabic or roman style. This is the *visuolinguistic* world. It contains objects - in this case icons - that exist in the real world, but have no direct spatiotemporal connection to their referents. The objects are just like sound utterances in verbal languages. From another order is the following package:

Definition 21. The iconicon is the set of all iconic concepts, without taking into consideration how they are depicted exactly. The term iconic concepts will be often abbreviated to icons in this report.

The iconicon is like a lexicon for verbal languages. It can also be compared with a standard like ASCII, that describes what the symbols should denote like "latin capital letter l", "euler constant" or "yen sign". It does however not describe how exactly they have to look like. It is possible to draw a cottage or a bungalow for the iconic concept "house". There is only one iconicon in the application VilAug. This is the *conceptual*[1] *world*. It contains abstract objects that have value because they refer to objects in the real world.

Definition 22. The (investigated) universe has three parallel worlds, the *unmodelled outerworld* with objects but also ideas and other observed phenomena is conceptualized and modelled in the *conceptual world* and gets concrete representamens, namely icons, in the *visuolinguistic world*.

This threefold division matches Peircean semiotics (see definition 18 on page 27), where the unmodelled outerworld contains the referents, the visuolinguistic world the representamens and the conceptual world the interpretants. See also figure 3.1 where the scene stands for the unmodelled outerworld, the icon sentence and word sentence for the visuolinguistic and linguistic world, and the persons in the form of brains with neural nets inside the conceptual world.

The icons in figure 3.1 are not from a real visual language, but some drafts about how such a sentence might look like. There is a mapping between the icon (or iconic concept) and the concept stored in the brain. It is not the same. It can however be the case that a verbal sentence like the Spanish sentence of above is less clear than the depicted icon sentence.

The iconicon is part of the visual language, an iconography is just a representation. Other parts of a visual language are besides the iconicon, the ontology and the grammar of the language. To know what the difference is between the iconicon and the ontology, we will further discuss these three components of a visual language in detail in subsection 3.1.2, 3.2.1 and 3.1.4.

[1] *or definitional or epistemological world*

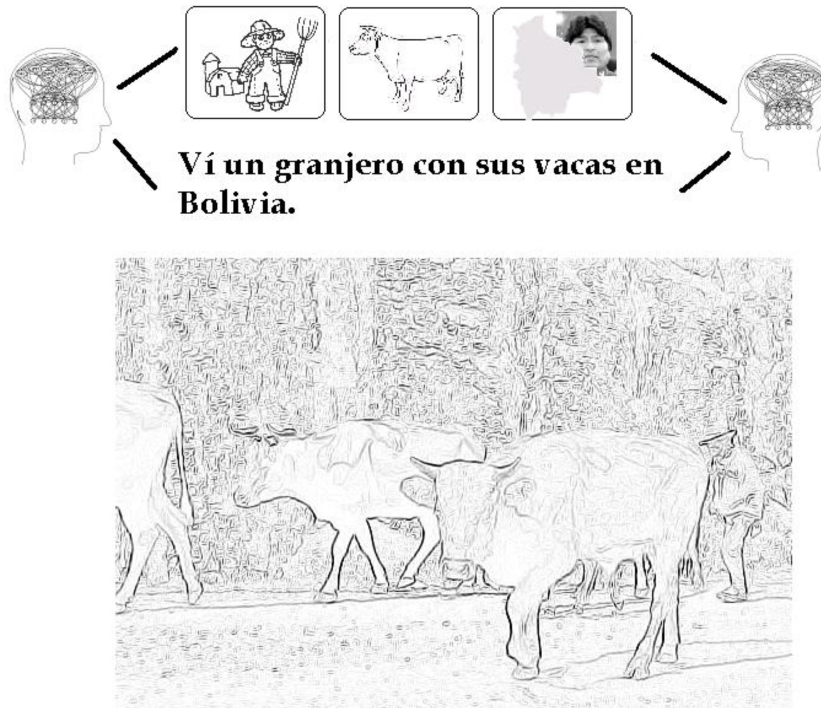


Figure 3.1: Three Parallel Worlds

3.1.2 Iconicon

The iconic concepts in the iconicon have several properties, called attributes. Like is shown in equation 3.1.

$$I = iconic\,concept(id, file\,reference, tooltip) \tag{3.1}$$

An iconic concept is unique by its identifier *id*. This identifier might be a string, but a number is preferred. There should be no verbal label that *identifies* an iconic concept. The motivation for this is that verbocentrism has to be guarded against. The inspiration of the visual linguist might come from gestures in a sign language. The icons might be based upon such signs. The visual linguist should not be forced to invent a verbal description for the iconic concept in that case. This, because the danger exists that accidently some linguistic bias accidently will contaminate the process (see about verbocentrism assertion 14).

The *tooltip* is such a verbal description of the iconic concept, but is not an identifier. The visual linguist is free to use many lines to explain what her icon means. That would be enough to describe for example a gesture in many details. The tooltip remains an approximation. An advanced reader or writer of a visual language, would not need tooltips for her icons anymore.

The $file\,reference$ is a reference to a file that contains a picture. It may have a file extension, but it does not contain a reference to the directory where the

VilAug IconMessenger IconNet

icon is stored. This is done on purpose and not a mere implementation detail. Icons can be ordered in separate folders, each folder delivers another type of representation. A folder can contain the icons designed by a specific graphic designer, or icons that obey specific or general design guideline principles like a black-white style or an urban style, or icons can be of different sizes. In this way is definition 22 established.

3.1.3 Ontology

Besides the iconicon, does a visual language also contain an ontology. With an ontology a lightweight ontology is meant (like in 2.2.3.3). It contains contain iconic concepts, and its therefore sometimes called *iconology* in this report. But it also contains relationships between concepts. These relationships stem from a particular context. They stand for the relation between the menu items on different levels (as visualized in figure 3.2).

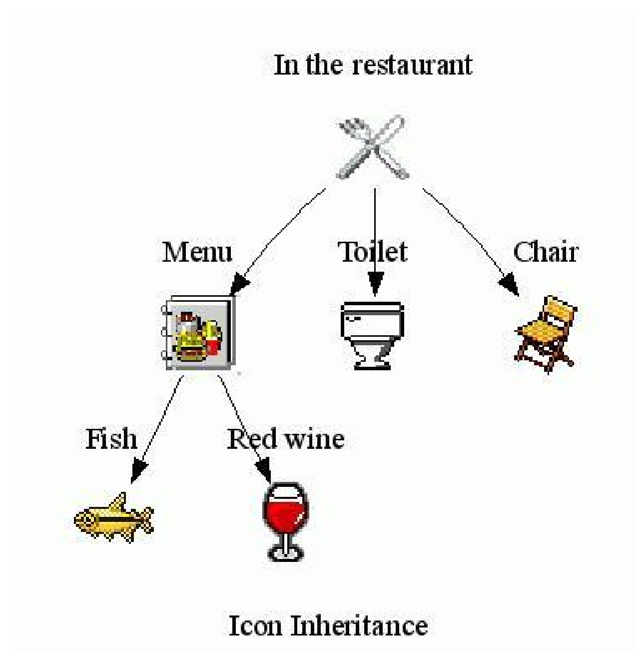


Figure 3.2: Example of Menu Level Division

This is the very important context of the nature of a relationship in the ontology, hence we will restate its use as follows:

Fact 23. *A relationship in the iconology (icon ontology) is a relation between a general icon and its derived, specialized icons like items in a menu or folders in a treeview.*

3.1.3.1 What's in a relation? Subsumption!

To explore the exact nature of this relationship we have to rely a little bit more upon mathematics. Consider the ontology as set C. The way icons are related in

figure 3.2 can be seen as a binary relation defined upon the set C. Such a binary relation connects multiple icons with each other, so it can be seen as a "specific manner to relate", abbreviated to R. The remark "using R" can thus be read as "using manner R (like the metonymy relationship) to relate". This relation is called the *strict*[2] *subsumption* relation and has the following properties:

- Irreflexive, a concept is not related (using R) to itself;
- Asymmetric, if concept $c_1$is related with concept c_2(using R) then is $c_2$not related with concept c_1(using R);
- Transitive, if concept $c_1$is related with concept c_2(using R) and is $c_2$is related with concept c_3(using R), then is concept $c_1$is related with concept c_3(using R).

The subsumption relation is less expressive than in other ontologies, where concepts are often reflexive. The motivation for its irreflexivity and assymetry is directly related to fact 23, where someone does not want to have infinite loops in descending the menu.

The way the properties of the relation are described have to do with the conception that relations are not merely relating concepts with each other, but are also concepts - namely relational concepts - themselves. See subsection 3.1.3.4.

3.1.3.2 What's in a relation? Conformity!

There is also another important relation in an ontology, the *conformity* relation, so points Nguyen [35] out. The conformity relation tells us that a certain individual conforms to a certain type or class. In "John is a man" the behaviour of John conforms to the features or characteristics of a man. In the iconology *itself* there is no difference between types and instances as such. In the ontology an iconic concept for "man" and an iconic concept for "John" can be created, and related by subsumption. There is no *internal* conformity relation.

The idea behind conformity is substitutability: "man" may be substituted for "John". There is another facet of the model that reflects this substitutability. It will be called the *external* conformity relation. This is the already mentioned separation between an iconic concept and its actual presentations on a physical location at the harddrive.

Definition 24. The conformity relation relates instances with types. The *internal conformity* relation relates instances with types in the same ontology. The instances are also described or identified in the ontology. The *external conformity* relation weakly couples instances of an ontological different nature (like real-world speech or icons on a harddrive) with types in the ontology. The type may only have a pointer to the whole set of its instances.

The actual constraints that a certain type applies to its instances are not prescribed. There is no "signature" involved in the mapping between concept and instance. Currently there are even no informal constraints, but they could exist in the form of (graphical) guidelines or a certain amount of votes in a webcommunity that indicates a picture's representational quality.

[2] *the ordinary subsumption relation can also subsume itself, it is reflexive*

VilAug IconMessenger IconNet

Assumption 25. Type and instance can not be distinguished in the same canon (collection of ontologies) in a consistent way across all imaginable specifications of conceptualizations (domain ontologies).

A consequence of the indifference between type and instance *within* the ontology, is that "concept" and "concept type" means the same. There is however still the external conformity relation.

Assumption 26. Type and instance can be distinguished if they have a different ontological nature. Types can be stored in a database table, while instances are books in the library. Types can be elements in an ontology, while instances are icons stored on the disk.

In conceptual graphs there are individual markers or one generic marker attached concept nodes. The generic marker (an asterisks $*$) is used to refer to an unspecified entity. The individual markers refer to individuals. In our model it can be the case that there is not even one representation for a certain concept. In this cases, a red cross will be visualized. This can be seen as a kind of generic marker in a slightly different sense.

3.1.3.3 What's in a relation? Signature!

The third relation we discuss briefly is the mapping between a relation like "hyponymy" and the concepts it has as its arguments. This involves a "signature" in an even more outspoken sense (than above). Each relation needs certain arguments. The amount of arguments is important, as well as the order of the arguments, as well as the type of the arguments. A relation from a certain type (defined by its signature) can be seen as a subtype of another relation (that has a little bit different signature). Nguyen and Corbett call this signature mapping among types the canonical basis function.[3] The arguments form restrictions in the way relations can subsume each other. An example given by them:

Example 27. Characteristic (Entity, Attribute) > TransportCharacteristic (Transport, TransportAttribute)

This means that the relation between transport and a transport attribute is a special case of the relation between entity and attribute. This is visualized in figure 3.3.

3.1.3.4 What's in a relation? A concept!

Figure 3.3 about the inheritance between relations, leads to the idea that a relation is just like another concept. A relational concept, but nevertheless a concept. It is connected to other concepts, it is part of a hierarchy, just like other concepts. Is "characteristic" a relation or can we see it as just a concept like others. A relational concept is more explicit in regard to the concept it needs as arguments. However, even a concept like "house" is clarified by related concepts like "to dwell", "family", "serves as a shelter" and "to live together". And is a "verb" a concept or a relation? As described in subsection 2.2 that

[3]They use also a "signature" function, but that function applies to relation nodes. While the canonical basis function applies to relation types. There is no such difference maintained over here, therefore the relabelling to "signature" (and later on the abbreviation "sign").

VilAug IconMessenger IconNet

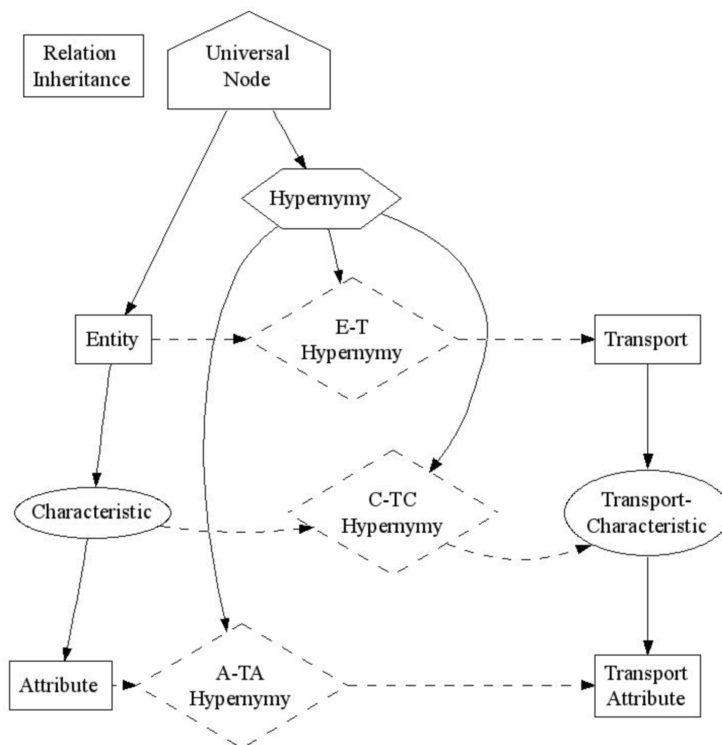


Figure 3.3: Relation Inheritance

briefly touches upon case grammar with its case frames for verbs. A verb can in this way be connected to several entities. As if it is a relation! This leads us to an important characteristic of the model:

Fact 28. *A relation should be treated as a first-class citizen just as a "normal" concept. Moreover, a relation is a relational concept and there are no reasons to regard it is an "higher-order" abstraction. The consequence of reviewing the relation as a relational concept in the context of the iconology is that a relation is also an iconic concept, and should have icon representations.*

The explicit removal of any unwanted discrimination between concepts and relations is very easy because the iconology forces us to make continuously a difference between the concept and its representation. A verbal description along the edges between icons in an ontology is very quick revealed as verbocentrism (assumption 14). Moreover, the conceptual graph theory (see also subsection 2.2.3.2) and an ontology like CYC (see subsection 2.2.3.4) already tend to go in that direction.

Also another formalism like Formal Concept Analysis [FCA] theory, does not explicitely distinguish between relation type and concept type. They are both called formal attributes. As Nguyen points out [35] also in Conceptual Graph [CG] theory it is difficult to conceptualize this distinction itself:

> "one of the current difficulties of the CG theory [...] is how to determine when something (a new idea) should be defined as a concept

VilAug
IconMessenger IconNet

> or should be defined as a relation (in the CG formalism). I hope that in the future, there will be a common agreement, a standard or some agreed guidelines by the CG community on this issue."

This shows that the distinction between a concept and a relation is not that clear-cut as may be thought. And in this report this is what with theorem 28 is formalized.

A manner to create relations with a sound hierarchy delivers for example Relational Concept Analysis. In this report by Priss [36] is the meronymy relationship in WordNet scrutinized. Handling relations as first-class citizens desires greatest care. Other components in a visual language are grammatical concepts. They have to be treated as first-class citizens too. How relations and grammatical concepts are related to each other will be discussed in the subsection 3.1.4. Let us first summarize however our findings in an encompassing datastructure, stipulate the pros and cons of the graph representation and add some general remarks about the iconology.

3.1.3.5 The canon

The framework holds ontologies for several languages. The entire collection of all ontologies, let they be general or domain specific, will be called the canon. Don't confuse the canon with a foundation ontology. There is no ultimate way of conceptualization proposed in this report. The canon can be formulated in the following way:

$$K = (T, I, subs, conf, sign) \tag{3.2}$$

In equation 3.2 is the set T a combination of two sets, namely T_C - the set of concepts - and T_R- the set of relations. These two sets are disjunctive and do not contain duplicate elements. The set I is the set of instances (or markers), concrete: the collection of images on the disk. The element $*$ is the generic marker that is associated by default with any concept and relation. The function *subs* is the subtype or subsumption relation like discussed: the strict subsumption $<$ instead of the common subsumption $\leq$. The function $conf$ is the relation between a type and a marker. This is in this case a weak coupling. Which instance is coupled to which type depends on the "working directory". Currently it can be the case that the same marker (except of course the generic marker) is used for multiple concepts. A unique mapping can however be enforced if its desirable for consistency. And last but not least the *sign* function, that involves the signature of relations in regard to their concept arguments. In our canon no explicit restrictions in terms of arguments are defined for the subsumption process. The *sign* function is therefore vacuous until now, it always has arity 3. Each relation needs a representamen, a referent and an interpretant. There are however no reasons to keep it that way.

Figure 3.3 shows - additionally - another facet of the nature of the canon. The set of concepts and the set of relations are not kept separated as such, but integrated in one hierarchy. This is also possible because there is no internal difference between relation types and relation instances.

3.1.3.6 Graph Operations

The subsumption relation can be taken to a higher level and generalized over graphs. A graph can be considered to subsume another graph. Over such graphs certain specialization functions (also called canonical formation rules) can be formulated as done by Chein and Mugnier (in [37]). The specialization functions are adapted significantly, but narrow down to:

- Simplification: delete a duplicate relation;
- Concept restriction: change a certain concept to a more specific meaning;
- Relation restriction: change a certain relation to a more specific meaning;
- Join: merge two vertices with the same description;
- Disjoint sum: merge two disjoint graphs by juxtaposition.

These graph operations are more or less the formal operations that can be defined over a graph. More specific versions and additional operations that are tailored to visual languages, are discussed later in subsection 3.2.3.

3.1.3.7 Model Capabilities

What is actually the motivation for creating an ontology anew, instead of using an existing ontology? There are several reasons for this:

- The ontology should be independent of verbal constructs, not only independent of words, but also independent of strings;
- The subsumption relation should not be restricted to the hypernymy, holonymy or antonymy type;
- The subsumption relation should not be undefined, a custom defined type should always be attached. An undefined relation would mean that an icon has always a particular child (in every language);
- The subsumption relation should be strict (irreflexive) because else an icon appears as its own child in the icon menu.

The relation of the model with a graph does also have its benefits. The specialization functions from subsection 3.1.3.6 come in a natural way, if the ontology is presented as a graph. But, there are more benefits of a graph presentation, that also appear quite naturally:

- The constraint from general to specific items is applied automatically by using a directed acyclic graph. Cycles can be prevented in an easy way. So, the ordering of general to specific icons is based upon previous conceptualized knowledge instead of predefined measures;
- Algorithmic ideas can be explored, that involves notions like path and neighbourhood. The distance between concepts can give an indication about meaning;

VilAug IconMessenger IconNet

- Graph theory, proofs and mechanisms can be extrapolated to this specific application.

Of course, our model does also have its limitations. To get a balanced view, read the next paragraph.

3.1.3.8 Model Scope & Limitations

It is important to note overhere that the ontology is not meant for reasoning. The graph rewriting mechanisms can be seen as merging different conceptualizations, not as manipulating knowledge statements about the world. This is the reason that there are no existential quantifiers. The ontology should neither be seen as a structure upon which a query can be applied (in the form of a specialization function) and the result constitutes the answer. This is by the way another reason why a distinction between type and instance is not meaningful in this ontology. However, for reasons as merging several ontologies, removing redudant information and appending conceptual information, the specialization functions of above fit the picture. Although the ontology does not contain factual knowledge, it still contains ontological knowledge. Still in other words, the graph is a representation of a subset of the canon, and not a conceptual graph. It is not allowed to have duplicate elements in the graph.

The limitations of the current model are the following; points mentioned above are repeated in a brief manner:

- The ontology is a lightweight ontology and contains no reasoning module (in the form of default inference rules);

- The ontology is a merge of different conceptualizations and the mapping between concepts in several languages is not automated (only the same identifier is recognized automatically);

- The ontology is not particularly aimed at machine processing, but is (only) a semantic web in the sense of a linked database for the general domain. There is no automation of corpus analysis (there is no corpus) or user analysis. The ontology data can be used in multiple application using a GraphML API (see subsection 4.5.1 on page 89);

- The ontology does not have comprehension of verbal texts as its purpose. Translations from a visual iconic language to a verbal or sign language can be done in several ways, but is not implemented. It is possible to use icon descriptions, to analyse the graphical properties of the icons itself, to investigate the relations with other icons, to analyse the way icons are used in a certain context, for machine translation. None of these translations that involve comprehension of the visual iconic language by the computer is added to the framework.

- The ontology is not converted to another ontology or knowledge representation format. Like for example the Conceptual Graph Interchange Format [CGIF], the Knowledge Interchange Format [KIF] or WordNet File Format [5WN]. The representation format has a GraphML format defining several GraphML attributes as additional data carriers;

VilAug IconMessenger IconNet

- The ontology is not based on direct recognization of the unmodelled outerworld. Like for example automatical icon acquisition from a scene, movie or photograph. Or generalizations of keyword queries on icons with search engines on the web. Or a formal analysis of visual properties of real-world objects. Linking the visuolinguistic world with the unmodelled outerworld is done by visual linguistics in several conceptually different ways;

- The ontology is based upon a graph representation, but no algorithms using the distance between two concepts or other metrics that might help conceptual recognization are defined;

- The relations in the ontology are not connected to concept arguments by a general "Canonical Basis function". Each relation is restricted to one representamen, referent and interpretant in the current implementation. The arity of a relation is three, its argument order fixed;

- The set of concept types contains only a supremum $\top$ (the universal type) and no infinum $\bot$ (the absurd type). So mathematically is the ontology a semilattice and not a lattice, There are many fine-grained concepts and an infinum would mean that many additional edges had to be drawn;

- The ontology contains only relations that make sense in the context of an icon ontology (strict subsumption). Therefore no relation that indicates negation, nor an inverse relation is defined. All relations link general concepts with more specific concepts;

- The ontology contains no existential quantifiers. This is considered part of reasoning about the ontology. Neither is conjunction and formation of sentences captured in the ontology itself. Neither is a sense of time or context-dependency in a corpus embedded;

- The ontology does not have a fixed set of unique beginners or top-level entities (except for the icons grammars and ontologies). This enables the use of several upper level ontologies in parallel;

- The concepts and relations are not explained by much more than their description and the connections to other concepts and relations. There are no verbal descriptions in several languages, no explanations in sign languages, no speech, no photographs, no animations, no movies, no ordering in explanatory groupings;

- The ontology does not contain an example of grammatical rules that apply semantic restrictions. For example that doubt (and its subtypes) can only occur together with human (and its subtypes) in one sentence.

- The grammatical parser only handles context-free and regular languages. It is not able to handle context-dependent or recursive grammars from the Chomskyan grammar hierarchy;

- The grammatical parser regards rules as including possibilities. It is not possible to exclude certain grammatical combinations except by including all combinations that are allowed. Like for example disallowing the combination of abstract object and colour in one sentence.

VilAug IconMessenger IconNet

These limitations are not all characteristic for the model. Some functionality is however excluded on purpose. Adding an absurd type is no problem if it is desired for mathematically soundness. The ontology is restricted to the context of an icon menu that may seem to lead to artificial constraints like exclusion of negation. Adaptions in this case are sometimes possible but should be grounded in the model.

3.1.3.9 Solution-Based Graph Operations

The mathematics for the iconology is rudimentary. The defined graph operations are described in a descriptive manner. They could have been formalized a lot more. The graph operation are described in a way that makes sense in the ontology. So, on creation of a relation a representamen, referent and interpretant have to be defined. A mathematical or logical equivalent of these transformation and specialization rules will not be given in this report. The rules as defined are convenient for the visual linguist. They reduce the danger of inconsistency significantly. Figure 3.4 shows this (although originally with a slightly different purpose).

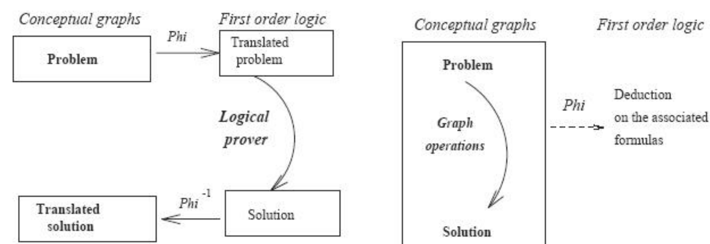


Figure 3.4: Solution-based Graph Rewriting (from Chein & Mugnier, [37])

With this we come at the end of our exploration of the iconology. No attention is given to *support* (or a support graph) that can enhance the iconology with information from other modalities, like photographs, descriptions in several languages, speech, music. Neither is attention paid to the issue of how to populate the iconology in an automatic way, how to create corpora of iconic languages, how to derive user characteristics. The specific advantage of the proposed ontology lays in the fact that it evolves naturally from the visual linguistic domain and brings under words matters like the blurrification between type and instance as described in assumption 25.

3.1.3.10 Historical Progress

Before we go on some more words will be said about the process of the iconology creation. The most important feature of the iconology would be that it had to store several languages from several developers. This lead the following corallories:

- Reusability of previously defined concepts as such (a predefined class “noun” should be reusable in several languages);

- Reusability of previously defined concepts as (sub- and super)types (a predefined class “noun” should be reusable as subtype of “instrument” in another language);
- Reusability of previously defined relations as such (an IS-A relationship can be used in many languages) ;
- Reusability of previously defined relations as (sub- and super)types (a custom relationship should be able to extend a default relationship);

The recognition followed that inheritance of relationships is easified by seeing them as concepts themselves and treating them as nodes in a graph. The latter appeared to be the case in conceptual graphs. This graph rewriting technique I labelled *ednoversion* (edge to node conversion). It is a kind of *reification* (making concrete that what was abstract until now). See figure 3.5 for an example.

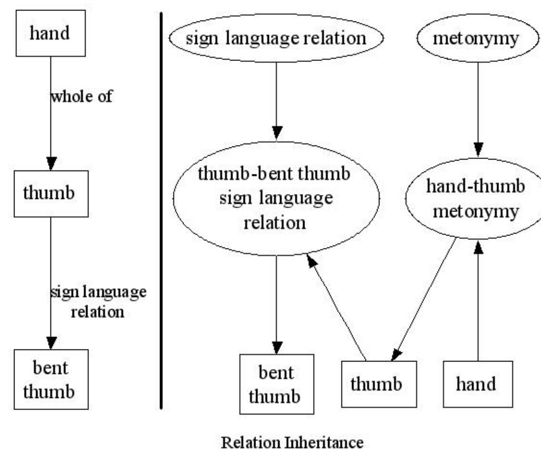


Figure 3.5: Ednoversion Example

The main difference with conceptual graphs is (besides the non-factual knowledge centered approach) the embedded type hierarchy in the ontology. With ednoversion the labels are recognized as concepts themselves and upgraded to nodes. In figure 3.5 the relation between “hand” and “thumb” is rewritten by formalizing the relation between them.

The example “John” is an instance of “man” is often used to indicate the difference between type and instance. It is very natural to depict “John” as an actual picture of that very person. The iconic concept “John” can even be coupled to different pictures or avatars of the real John. So, even this is not a real instance in the iconology. Therefore no strict separation between type and instance is maintained. The objects that are the terminals in the graph can in some way be seen as instances, but only in this remote way.

This report will continue with a third component of a visual language, namely its grammar. After that we return to the topics of this subsection - concepts and relationships - but now particularly in regard to visual languages. What datastructure responds to the idea that an icon is part of a language? Are there ways to structure concepts or use layers in a language that are hidden to

VilAug IconMessenger IconNet

the user and thus not reflected in the menu? These kind of questions remain to be answered.

3.1.4 Grammar

Besides an iconicon and an ontology a visual iconic language does contain a grammar. There is only slightly touched upon this issue in the discussed theory, in subsection 2.1.2.1 about language characteristics. The most important property of a language as considered in this report is that a grammar *restricts*. It disallows certain combinations of sentence elements. They have to appear in a certain order for example.

3.1.4.1 What's in a Grammar? A restrictive Criterium!

There is not only one type of grammar. Traditional grammar does use the "noun", "verb", "adjective" concepts. Above that adds grammatical sentence analysis concepts like "subject" and "object". Besides these well known parts of speech, other manners of assigning labels to sentence parts are possible as well. One of these is the already mentioned case grammar (see subsection 2.3.2). A case grammar uses semantic roles. Semantic roles like the "patient", "instrument" or "locative" role.

In our ontology it is important to reuse elements, also elements that are part of a grammar. So for the case of reusability, general themes between grammars have to be investigated. This is not to discover general intergrammatical truths or a universal grammar. This is to discover what concepts can be shared among different grammars. In this report the following is assumed:

Assumption 29. A grammar is distinct from others, because the criterium that selects its (grammatical) elements is different. That criterium can use inflection, order or other metrics to decide how to decompose a sentence in grammatical elements.

If this assumption turns out to be unfounded, this is not detrimental to the model. It has two consequences. Firstly, to stimulate creativity in considering other grammatical constructs than the existing ones. The latter are again and again swiftly tending towards verbocentrism (see assumption 14). Secondly, grammars that can be distinguished using such a criterium, will be implemented in the ontology. If they are inappropriate they can be adjusted or replaced.

3.1.4.2 Grammar Types

We have talked about the different types of grammar in an abstract sense. But what does it mean in a concrete manner? The following grammatical mechanisms are distinguished in this report:

- *Frequency criterium*: The concepts patient, instrument, locative are selected upon their frequency in a sentence (they should occur once in a sentence according to an axiom somewhere [38] called the Unique Case Principle);
- *Relationship criterium*: The concepts subject, object, oblique are selected upon their relationships in a sentence (SVO languages, elements come in phrases);

VilAug IconMessenger IconNet

- *Inflection criterium*: The concepts noun, verb, adjective, determiner are selected upon their form change abilities in a sentence (inflection, a noun can get a plural marker, an adjective gender inflection).

These three grammar types will be (partly) implemented. The given criterium names can appear throughout the documentation (in the form of for example "frequency grammar"). Grammars are in reality often combinations of the given criteria. Case grammar can for example also be seen as arising from "case inflection" (see Anderson [39]). And Gutiérrez (see 5.1.1.1 in [34]) describes even other critera that have to do with completeness, uniqueness, distinctness and independence. All kind of grammars can be created in practise and even combinations of the mentioned criteria can be used to compose a grammar.

3.1.4.3 Grammology and Grammicon

Practically, there are three problems to be solved regarding the grammar. Firstly, the visual linguist has to define which grammatical items exist in her grammar. Her criterium is allowed to be informal, but she should indicate which grammatical concepts like "patient", "instrument" or "subject", "object" exist in her grammar. Secondly the visual linguist has to define which iconic concept falls into which grammatical categories. Can "house" be categorized under "patient"? Can "sigaret" be categorized under "instrument"? Thirdly, grammatical rules have to be written that order the grammatical categories and collect them in (recursive) phrases. Other problems (like applying these rules) belong to the tasks of the application.

Like the set of icons is called the iconicon, so is the collection of grammatical items called the grammicon. And like the ontology that structures the set of icons is called the iconology, so is the ontology that structures the set of grammatical items called the grammology. The grammar rules do not receive a specific name. The relations between the grammicon and the iconicon are the topic of the next subsection.

3.1.4.4 What's a grammatical relation?

Like described in the previous paragraph, in a visual language are icons mapped upon grammatical categories. This will be meant with a "grammatical relationship" in this report. Is this relationship in any way different than the relationship as discussed in subsection 3.1.3.4? Is there something magic about classifying "bird" as a "noun" instead of an "animal"? This has to do with regarding grammatical relationships as first-class citizens of the ontology too, just as concepts, just as relational concepts. Hence assumption 30:

Assumption 30. A grammatical relation should be treated as a first-class citizen just as a "normal" relation. Moreover, a grammatical relation is a grammatical, relational concept and there are no reasons to regard it is an "higher-order" abstraction. The consequence of reviewing the grammatical relation as a grammatical, relational concept in the context of the iconology is that a grammatical relation is also an iconic concept, and should have icon representations.

Verbocentrism plays also a role in respect to grammar. It is not right to limit the visual linguist to traditional grammatical concepts like verb, noun, adjective.

The difference between the adjective SURPRISED and the verb SURPRISE does not arise naturally from a graphical point of view. It is also important to consider the post-construction situation. When these icons are displayed in a sentence in an icon document the difference should still be visible! The grammatical cases of the icon SURPRISE should be graphically distinguishable. Or grammatical cases that better suit the visual language should be used.

Remark 31. The characteristics of a visual language can be used in determining what kinds of grammatical elements but also what grammatical *features* can be applied. Consider for example quantifiers. From a graphical viewpoint a picture denoting VERY SURPRISED is better than a combination of the two icons VERY and SURPRISED (see Principle 3.22 of Leemans dissertation[2]). Also concepts like RED ROBE or MAN BEHIND CAR could be captured in the same icon.

We wil return now to already discussed material and review it in the light of a visual iconic language. The discussion of the grammar adds some additional questions. What datastructure correlates with the idea that an icon is part of a grammar? Which of the grammatical criteria can best be used for a visual language? How can the same type of grammar be used across different visual languages? We will answer these questions in the next subsection.

3.2 Applications & Examples of the Model

This subsection describes the application of the model with the visual languages it contains. It is not only a question about what components are needed, nor about an exact definition of the terms that are used, but especially a question about how the model components will interact. How will concepts, relationships and grammatical classes be combined towards a meaningful pattern?

3.2.1 Iconology

This subsection describes the model in which our requirements and assumptions will be translated to the domain of a visual language. What does it mean to have a relation between two icons in the iconology? How is this realized? This will not descent to questions regarding the implementation. It is still implementation independent. It does however clarify the kind of datastructures that have to be implemented. It is the last resort before delving into UML descriptions. Descriptions that describe what kinds of models and classes are needed, but don't leave room anymore for questioning their existence. So, let us investigate the questions like: "When is an icon part of a particular language?" and "how is information inherited?".

3.2.1.1 General Structure

Let us first establish the graphical difference between an ICON and a "description" of an icon. The layout ICON will be used to emphasize its iconic nature, without the need to use graphical items throughout this report. Until now most often a "description" of an icon is used. Try to mentally visualize in each case a real picture.

The several domain ontologies are each represented by an icon. One of these ontologies is LEEMANS' ONTOLOGY. This icon is represented by a combination

of the icon LEEMANS and the icon ONTOLOGY. This icon is a subclass of ONTOLOGIES. There is no intermediate relational icon between ONTOLOGIES and LEEMANS' ONTOLOGY[4]. Like already discussed the ontological classification as well as the grammatical classification makes extensively use of the classes "verb", "noun" and "adjective" in VIL. The first two are *unique beginners* (top-level entities) of Leemans' ontology. The icons VERB and NOUN are connected to LEEMANS' ONTOLOGY in the following way. The top of the iconology is the icon TOPOLOGY and unique beginners are attached to this top entity by an intermediate relating icon, in this context: TOPOLOGY-NOUN RELATION and TOPOLOGY-VERB RELATION. To indicate that an icon is a unique beginner of particulary Leemans' ontology, the icon LEEMANS' ONTOLOGY is connected to TOPOLOGY-NOUN RELATION. See figure 3.6.

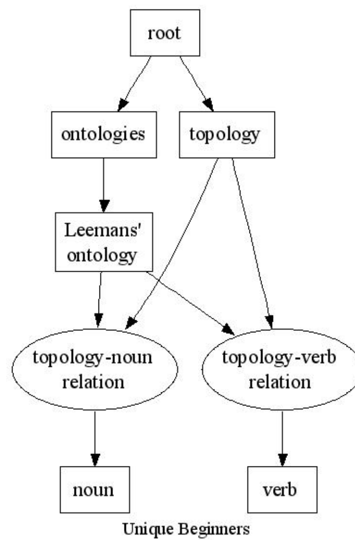


Figure 3.6: Unique Beginners in Leemans' ontology

This figure (3.6) is not really different from figure 3.3 or figure 3.5. Also Peirce notion of a sign can again be recognized. The "sign" TOPOLOGY-NOUN RELATION does have as referent the icon NOUN, as representamen LEEMANS' ONTOLOGY and as interpretant TOPOLOGY. The Peircean notion of a sign is a little bit adjusted in this report. It is considered a matter of fact that the roles of representamen and interpretant can be swapped. With TOPOLOGY as interpretant its sense has to be read as "hasUniqueBeginner(LeemansOntology,Noun)", with LEEMANS' ONTOLOGY as interpretant its sense has to be read as "inLeemansOntology(UniqueBeginners,Noun)". This interchangeability is important enough to state properly:

Fact 32. *A sign in the iconology contains three proponents like in Peircean semiotics: representamen, referent and interpretant. The main difference is the interchangeability of representamen and interpretant.*

[4]So, the canon is currently not a bipartite graph with between every conceptual node a relational node. Maybe this should be enforced.

VilAug IconMessenger IconNet

This will also be important in regard to grammar later on. This interchangeability establishes the direction of the edges in the iconology. The edge in direction to the referent has to be different from the others. The convention is that this edge will be pointed towards the referent, the other two edges are pointed from the representamen and from the interpretant towards the relating icon.

3.2.1.2 Inheritance of Icons

Inheritance in the concept of the iconology involves using icons and hierarchies of icons of already existing ontologies. This is necessary when someone wants to add additional, specialized icons as subtypes of existing icons. Suppose someone wants to define the icon HORSE in an ontology for a zoo. Suppose also that a biological ontology already exists. This ontology contains animals from the level of order to genus, including the icon EQUUS (to which horses, zebra's and donkeys belong). To incorpate the icon EQUUS in the ontology for the zoo, it is sufficient to relate it to an already existing icon in the ZOO ONTOLOGY, in figure 3.7 this is the top of the ontology, TOPOLOGY.

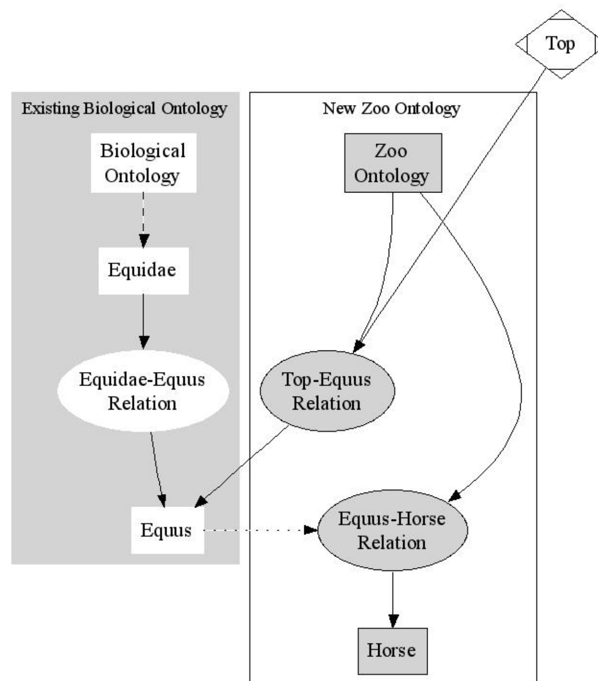


Figure 3.7: Icon Inheritance Example

It could have also been possible to relate the zoo ontology at one point higher in the biological ontology, to the EQUIDAE icon. This however does not automatically include the genus EQUUS to the zoo ontology. How such a whole hierarchy can be inherited is explained in the next subsection.

3.2.1.3 Inheritance of Icon Hierarchies

There are two approaches possible in regard to inheritance of icons across ontologies. Pretty straightforward is to add an additional edge between a normal concept and relational concept in two different ontologies. In figure 3.8 such an additional edge is added between the ZOO ONTOLOGY and the EQUIDAE-EQUUS RELATION. If iconic concepts surrounding the relation in the original ontology are connected to relations in the new ontology, this does have as result that also the relation in the original ontology will be obeyed in the new ontology. Figure 3.8 visualizes this. This lattice structure between two different ontologies shows how much two languages can be interwoven. How there is still a way to distinguish very reliable ontologies from less reliable ones, is solved by file modularity (see subsection 4.5.3).

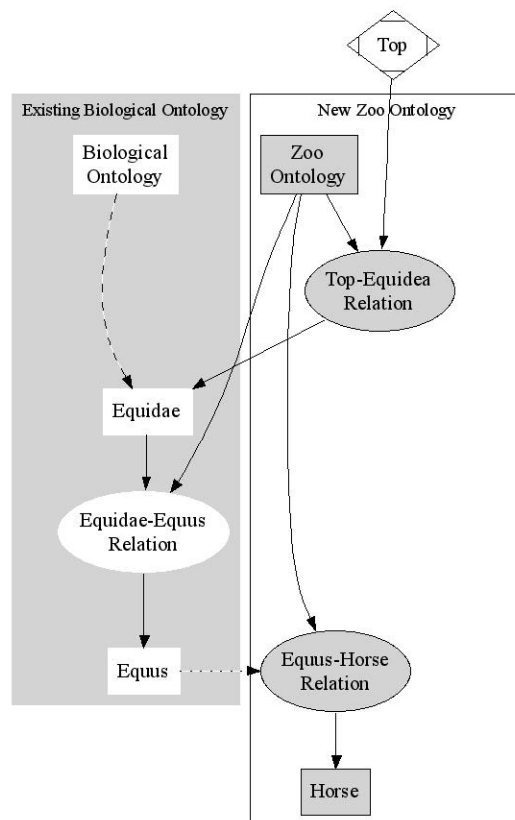


Figure 3.8: Weak Inheritance Example

This manner of inheritance is called *weak inheritance* because it uses a restrictive manner of adding previous existing datastructures. The icon EQUIDAE is added to the ontology, but that doesn't mean anything for hierarchies beneath that icon. Each of its subtypes has to be added manually by connecting them as shown in figure 3.8.

There is also a stronger way of inheritance, namely by adding a supertype of the EQUIDAE-EQUUS RELATION to the zoo ontology. As visualized in figure 3.9.

VilAug IconMessenger IconNet

This needs thinking through of the semantic relations someone wants to use in an ontology.

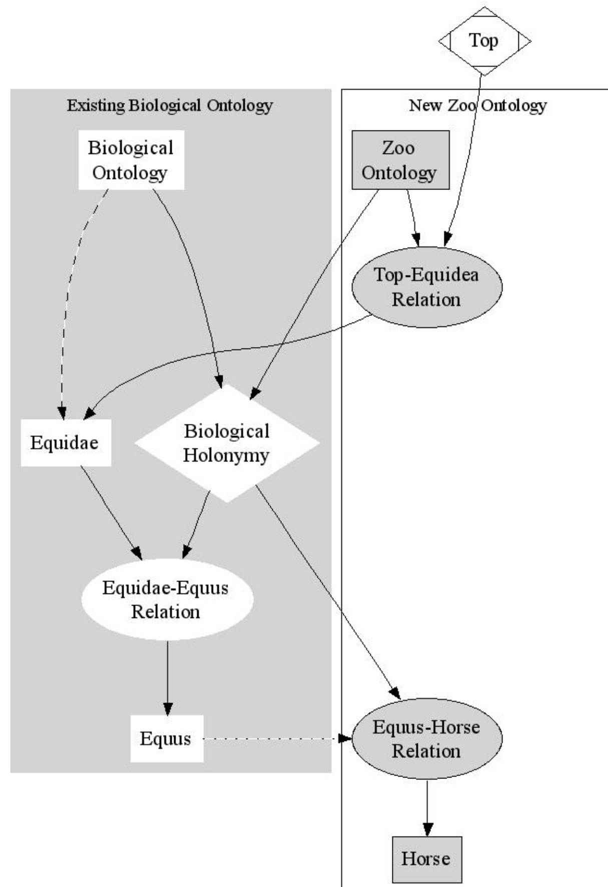


Figure 3.9: Strong Inheritance Example

This is considered as a *strong inheritance* because relations that are connected to BIOLOGICAL HOLONYMY are all at once part of the new zoo ontology. So if VERTEBRATES is added to the zoo ontology, all its subspecies are added too. It is therefore important to have a conceptual density across relational concepts. If there was only a HOLONYMY relation defined, the visual linguist had to create the substructure beneath BIOLOGICAL HOLONYMY herself. While that actually should be part of the biological ontology. This method of inhertance is not everything the iconology offers. It is imaginable that zoo ontology wants to inherit the subtypes of EQUUS, but not visualize the icon EQUUS itself. Is there a possibility to inherit structures without visualizing them? How structures can be hidden will be discussed in the next subsection.

3.2.1.4 Hidden Structures & Interfaces

Inheriting a datastructure and not visualizing its icons is a little bit contradictionary. However, there is a method provided in the iconology that offers this possibility. This method makes use of so called *interface* nodes. The creator of an ontology has to make such interface nodes available. These nodes function as anchors for inheritance. The nodes will not visualize because they are marked as interfaces. It is not allowed to “derive” an interface from a relational concept. It should always be preceded by a normal concept. So, these nodes come in triples: the interface node, a node that visualizes the concept, and one node that hides the concept. Let us again view the same example, see figure 3.10.

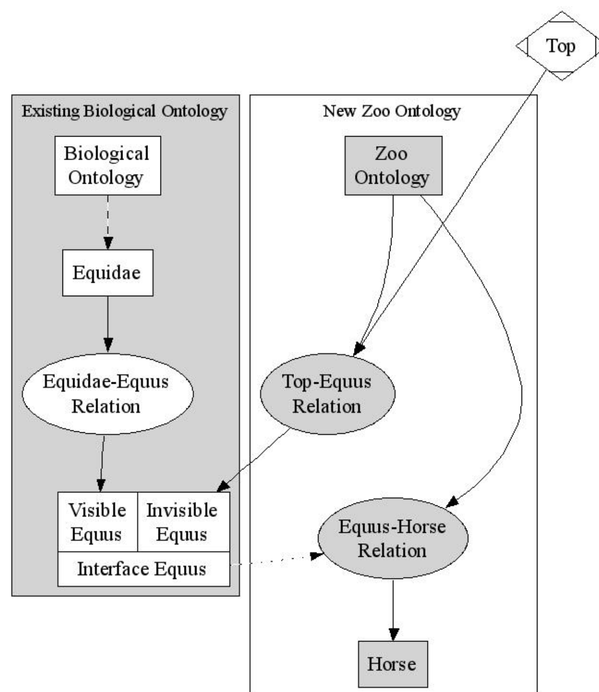


Figure 3.10: Interface Inheritance Example

In figure 3.10 it is shown how the biological and zoo ontology differ in inheriting from EQUUS. The biological ontology links to the visible EQUUS, the zoo ontology links to the invisible EQUUS. As unique beginners in the zoo ontology, the EQUUS icon will therefore not appear, but instead of that the icon HORSE. The information about visibility can not be stored on the concept itself, because it can be used by many ontologies. Another solution would be to add visibility information to the edge between TOP-EQUUS RELATION and EQUUS. In that way this would be the only need for semantics upon edges in the ontology. Until now we have not talked about algorithmic simplicity. What adding visibility information to edges implies, is that all the browsing algorithms also have to consider this visibility field upon every edge in that case.

Another point has to be made. The equality of iconic and relational concepts is emphasized until now, see theorem 28. However, there is of course a difference,

VilAug IconMessenger IconNet

namely the visibility of the relational concepts in the hierarchy, as defined in 23. They are *not* directly visible, but they are indirect part of the hierarchy just as the visible concepts. Slowly we are progressing to the end of this chapter about the model details. To know exactly how the ontological and the grammatical part of a language interact, we have to investigate a few additional situations in the following subsection.

3.2.2 Grammology

The way a certain icon is assigned to a certain grammatical category is just as it would be assigned to an ontology category. A grammatical ontology can be seen as a special type of a domain ontology. This is visualized in figure 3.11.

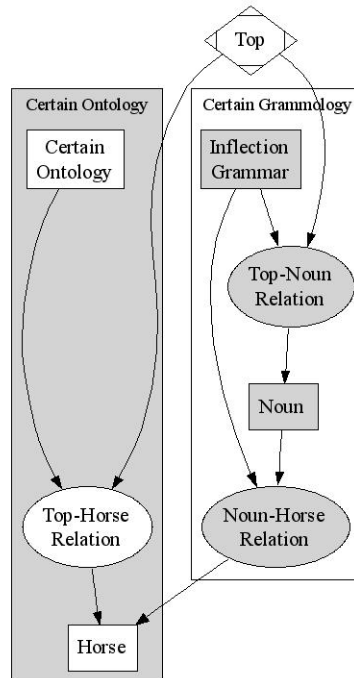


Figure 3.11: Grammatical Categorization Example

The icon HORSE is in figure 3.11 connected to the grammatical icon NOUN. And concepts like NOUN are defined in a normal way as in figure 3.6, only now in the context of a grammology. When a new icon like HORSE is added to a language, this involves summarized two steps. Firstly, connect it to its parent in the appropriate domain ontology, and secondly, connect it to its parent in the appropriate grammalogy. This example makes again clear that it is senseful to use concepts like NOUN over different ontologies, this concept is therefore presented as an interface (see previous subsection).

VilAug IconMessenger IconNet

3.2.3 Graph Mutations

Now we know a lot more about the actual datastructures and how they combine ontology and grammar, allow for inheritance of icons, relations and grammatical concepts, and how visibility of concepts can be adjusted. It is time to consider or actual reconsider the graph operations as described before in subsection 3.1.3.6 about graph operations. They will not be viewed as graph operations anymore, but as graph *mutations*. Imagine the user that has this ontology captured in a graph and wants to execute actions like described above. Subclassing a relationship, hiding an iconic concept, classifying an iconic concept under a grammatical category, etcetera. These things narrow down to executing mutations upon the underlying graph. The defined set of graph operations are therefore not sufficient anymore for our purposes. So, let us define an extended list of user actions, that can be seen as requirements for the implementation phase:

- *Redundant concepts removal*: remove duplicate vertices, merge two vertices with the same description (join), merge two vertices represented by the same picture;
- *Redundant relations removal*: delete a duplicate relation (simplification), a relation is a duplicate if it refers to exactly the same concepts (representamen, referent and interpretant are the same);
- *Irredundant concept removal*: removes a vertex (and corresponding datastructures like relations that are now lingering, see subsection 3.2.3.2);
- *Irredundant relation removal*: (like remove irredundant concept);
- *Concept refining*: refine a certain concept by adding subconcepts;
- *Relation refining*: refine a certain relation by adding subrelations;
- *Concept restriction*: change a certain concept to a more specific meaning (by changing its description, by adding an additional type between it and its supertype);
- *Relation restriction*: change a certain relation to a more specific meaning (by changing its description, by adding an additional type between it and its supertype);
- *Disjoint sum*: merge two disjoint graphs by juxtaposition (using operations of above);
- *Add concept*:: see concept refining;
- *Add relation*: add a relation between two concepts (by defining a representamen, referent and interpretant);
- *Upgrade relation*: upgrade a relation to a (normal) concept (by adding relations between the original relational concept and its representamen and referent, also called "activate icon");
- *Interface concept*: make an interface of a concept (see subsection 3.2.1.4).

VilAug IconMessenger IconNet

Some of these points need some additional clarification. Someone may ask what exactly will be removed in a "remove redundant concept" operation. This will be handled in subsection . The penultimate mutation "upgrade relation" will be explained in subsection 3.2.3.1. The "interface concept" mutation involves visibility in the resulting menu. The label "add visibleness" is also used for this mutation. It is not a matter of setting a visibility parameter, but adding the possibility to set a visibility parameter. Hence it is called "add visibleness" instead of "add visibility".

3.2.3.1 Upgrade Relation

What would be the reason to visualize a relational concept? An illustration of the sense of this procedure is given in figure 3.12.

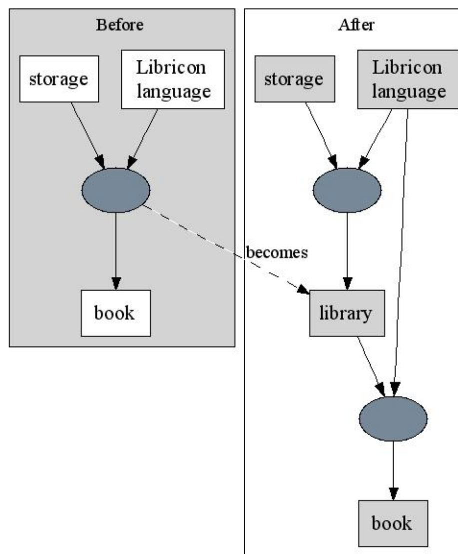


Figure 3.12: Visualization of Relational Concept Example

Suppose there is a language that contains the icons BOOK and STORAGE and a relationship between them. What would be more appropriate than specifying this relationship in some way or another? From the viewpoint of the builder of an ontology this can be seen as an awkward transformation. However, for the builder of the graph it can be intuitive. The linguist is able to assign icons to relational concepts (see theorem 28). The icon assigned to a relational concept can be seen by the linguist as a kind of "passive icon". In the case of above the relation between BOOK and STORAGE can be depicted by a COLLECTION OF BOOKS. This can motivate the linguist to make it a normal icon and hence the need for this mutation is born. The label "activate icon" can therefore also be used to describe this graph mutation.

3.2.3.2 Remove Concept

The mutation that deletes a concept comes in different cloths. See first figure 3.13 where a typical construct between iconic concepts and a relational iconic concept is depicted.

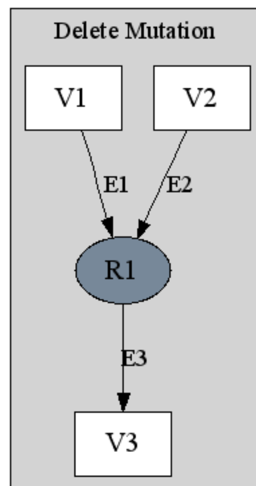


Figure 3.13: DeleteConceptMutator

Each of the elements in figure 3.13 might be removed. However, the removal procedure of vertex V1 is not different from removal of vertex V2. Three types of removal can be distinguished:

- Delete Iconic Concept R1: The edges E1, E2 and E3 will be deleted, and when V3 becomes isolated (no parents) it will be removed too;

- Delete Iconic Concept V3: The edge E3 will be deleted. When R1 does not point to other vertices, R1, E1 and E2 will be deleted too;

- Delete Iconic Concept V1: The vertex R1 and edges E1, E2 and E3 will be deleted. When V3 becomes isolated, it will be removed too.

Another possibility (on deletion of concept V1) would have been to try to relate V3 with superconcepts of V1. This would make from {SPORT & TYPE -> TO SWIM} and {TYPE & TO SWIM -> BORST CRAWL} , the new construct {SPORT & TYPE -> BORST CRAWL}. This makes sense in the case of particular parents like TYPE. However, inference belongs not the topic of this knowledge representation system (see subsection 3.1.3.8). Besides, when it would be implemented it should also define how to build a construct from {SPORT & TYPE -> TO SWIM} and {ENVIRONMENT & TO SWIM -> WATER}. In that case a construct like {SPORT & TYPE -> WATER} does not make sense, while a {SPORT & ENVIRONMENT TYPE -> WATER} does make sense.

VilAug
IconMessenger IconNet

3.3 Model Requirements

The list of graph mutations or user actions (of previous subsection 3.2.3) that require implementation is a good starting point to consider what kind of functionality the theoretical model desires. These mutations are the first requirements for the to be developed software model. They will not be repeated overhere. There are also additional requirements in terms of the datastructure that in some way are not emphasized until now. And also an important topic that is not discussed until now, is the way the grammar is parsed.

3.3.1 User Actions

The user actions that have to be possible are listed in subsection 3.2.3. So the user should be able to remove redundant concepts and relations, refine concepts and relations, restrict concepts and relations, add concepts and relations, upgrade relations and interface concepts. These actions can be performed by using some kind of command line script. It is however recommended to use *direct manipulation* in this case. This entails visualizing the graph and enabling the user to drag and drop vertices and relations. These requirements should lead to an application that is really useful in practise, also for the domain expert, the visual linguist.

3.3.2 Data Structure

Another restriction has as purpose to stimulate the visual linguist to use existing relationships and embed them in a generic to specific way. This graph that should be created is therefore a directed acyclic graph [DAG]. The acyclicity forbids the developer to create relations that involves infinite loops in the icon menu of the final user. It also reduces "reciprocal information". A lot of redundant information - coined *reciprocal information* - is stored if a developer decides to have hyponymy nodes exactly parallel with all meronymy nodes. That is why this acyclic restriction is added to the datastructure. This constraint should be relieved if it turns out to be too restrictive. Another restriction involves multiple edges. Edges themselves do not carry any semantics. There are no edge labels. Only the fact that eges connect nodes, is of informational value. This means that multiple edges are not necessary (and can be forbidden).

3.3.3 Parsing Grammar

There are many possibilities to parse grammars. However, there are also many requirements that our visual language parser has to fulfill. This will limit our choices considerably. First of all the parser has to be an *interpreting* parser. That means that it operates simultaneously upon a grammar file and a source file (with icons) and is able to apply the grammatical rules directly upon the icons in the source file. Nothing of the grammar can therefore be built explicitely in the parser. So parser generators do not pass this test. They take an arbitrary grammar and create runnable source code as output. Subsequently this code has to be incorporated in the application. That is one bridge too far, and parser generators can therefore not be used.

Second property is that it has to be *sequential* parser. The sentences that the parser operates upon, are not fixed in advance. This is different from many programming languages, in which the parser only executes its task considering the whole document. Every unfinished sentence means an error in that case. This parser however should be able to have symbols added to it sequentially. It definitely should not cause an error when receiving the icons one by one.

Third pecularity is that it has to be a *predicting* parser. This is directly related to its sequential character. Because the amount of images that can follow a certain sequence is limited, it is very well possible to list only the correct follow-ups for a sequence. This forces the user to produce correct grammatical sentences only.

Fourthly, it has to be an *ambiguous* parser. The symbols that are used as input can not always be tied to only one grammatical category. When an icon can be a "noun" or a "verb" the parser should remain agnostic until additional icons reveal further clues. By the way, this type of ambiguity is different from having multiple parsing trees (what is called a forest) resulting from one sentence. The situation is even worse: It are several of these forests!

3.3.3.1 Order

There are additional characteristics of the parser that do not have to do much with the grammar type. One of such an additional requirements is the way order is handled. There are *ordered grammars* and *unordered grammars*. The grammar of Fitrianie is order-dependent, the grammar of Leemans is order-independent. Both grammars should be formulated as BNF rules in an XML format. The apparent order of an unordered grammar should be disregarded by the parser. Somewhere the parser does already generate two rules in stead of one when encountering an optional symbol. This would also perfectly fit with adding other permutations like offering the sentence in different order. This will delay the parsing considerably. However, there shouldn't be many elements in an unordered grammar, because everything will result in one big rule anyway. Only the frequency of elements differs in such grammar.

3.3.3.2 Automatic Grammar Classification

There are (at least) two approaches regarding how to actually impose grammatical restrictions. These two approaches use different kinds of visualizations. The first technique presents the grammatical concepts directly to the user. The user starts navigating the icon menu by choosing a grammatical category that is subsequently refined by icons that belong to this grammatical category. The grammatical concepts are in this case the unique beginners or roots of the ontology. Every icon in the ontology should be subtypes of these grammatical concepts. This technique will be called *user grammar-classification*.

The other technique is called *automatic grammar-classification*. This technique derives automatically the right grammatical category when a certain icon is chosen. This automation can be of different orders. It is possible to relate an icon like for example THOUGHT to the (grammatical) parent NOUN and the parent VERB. With automatic grammar-classification there is no indication of which grammatical parent is appropriate. The parser can automate this process by looking at the grammatical roles of the other icons in the sentence. Stochas-

VilAug
IconMessenger IconNet

tics about commonly entered sequences is useful, as are human preferences, local preferences, personal preferences, iconic language pecularities, etcetera. A corpus to invest such characteristics of a visual language does not exist.

No requirement for automatic grammar-classification will be imposed upon the software model.

VilAug
IconMessenger IconNet

Chapter 4

Software Model

The software model, also called architecture or framework, contains topics that belong to the domain of the programmer. The packages are described on the level of UML diagrams. The main modules, the IconModule, the GrammarModule, the IOManager are modelled. A specific grammar parser (Earley parser) is chosen and its components modelled. The syntax, or API for datatransfer is established in GraphML. Browsing strategies for the graph structure will pass the examination. And the technique to filter vertices for the graph is discussed.

The content of this chapter should allow someone to develop an application that has in general the described functionality. This can be in another language or upon another platform. However, the application uses a large graphical library and many modelled objects are extensions upon this library. An an equivalent library would be needed in such a case.

4.1 Context

In general we know what functionality our software model has to contain. However, the context of the problem is until now not sketched in much detail. The field of visual linguistics does yet not exist. The job of the visual linguist does not exist either. With what kind of people should a visual linguist coorporate? What kind of tasks should a visual linguist have to perform? There is yet no corpus of visual languages. Analyses of corpora is therefore impossible. Tasks in regard to defining and describing visual languages would however be part of her daily routine. The context in the form of an extended stakeholder diagram is given in subsection 4.1.1 and the supposed tasks the software model should be able to comply is given in subsection 4.1.3.

4.1.1 Stakeholder Involvement Diagram

It is important to be aware of all the parties that can play a role in the development and use of this framework for visual iconic languages. Therefore a kind of stakeholder involvement diagram is given in figure 4.1 on page 73. This diagram depicts possible use cases surrounding the intentional product. It is a stakeholder diagram expanded with use cases. The actors that provide input are depicted at the top, the actors receiving output at the bottom. It gives a

graphical overview of the context of the problem situation.

Figure 4.1 shows a lot of use cases in the form of ellipses. The ones at the center that start with the verb "Manage" are main activities in the proposed framework. Manage Iconology is about the (icon) ontology and its routines. Manage Iconicon represents and stores iconic concepts. Manage Icon Grammar is about grammatical concepts and rules. Manage Machine Formats provides interoperability between applications. Manage Iconography handles pictures and fonts. And Manage Icon Representations regards representations, translations and visualization methods to present the icons in other forms or explaining context.

The software design will especially emphasize the Manage Iconology use case (see section 3). Not everything in this figure will be implemented, nor designed. Like a tool to create grammatical rules with a GUI. Or tools that aid the creation and design of pictures itself. Tools that can be added to the framework will be discussed in section 7.6 on page 148 that contains recommendations.

4.1.2 System Overview

An overview of the components involved is given in figure 4.2 on page 74. The provided artifacts of the entire process are in the form of XML and GraphML data files. Like can be seen in comparison with the stakeholder involvement diagram in figure 4.1 on the next page will only the components around the iconology be modelled in this stage of development.

The users are at the left side of figure 4.2 on page 74 and provide input to the system. They communicate with certain tools in the framework, labelled IconNet (see subsection 4.2.3) and IconMessenger (see subsection 4.2.2). These tools are connected to the iconology (see previous subsections 3.1.3 and 3.2.1 about the ontology). This iconology contains or is connected to an iconicon (lexicon of iconic concepts), grammicon (lexicon of grammatical items) and grammar rules. The actual pictures are stored in a picturebase, an iconography. Besides output with visualization and communication purposes for the users, there are APIs defined to enable interoperability between machines (and applications).

What kind of functions are embedded in the framework components will become clear after the task requirements are transformed to system functions (see subsection 4.1.3).

4.1.3 Task Requirements to System Functionality

Certain procedures that have to do with updating and changing the iconology are already described in subsection 6.3 on page 127 in section 3.3 on page 68 about the model requirements. That subsection refers to the previous subsection 3.2.3 that lists user actions (called graph mutations) in regard to the iconology. An extended set of these user actions will be described in this section.

Figure 4.3 on page 75 displays the tasks the user wants to perform at one side of the figure, and the tasks the system has to perform at the other side. So figure 4.3 maps user task requirements to system functionality.

In figure 4.3 some new requirements appear (besides the ones listed are already listed in subsection 3.2.3). First thing that catches the attention is the fact that two types of users have a prominent role. The visual linguist and the instant messenger user. The tasks a visual linguist should be able to perform

Figure 4.1: Stakeholder Involvement Diagram

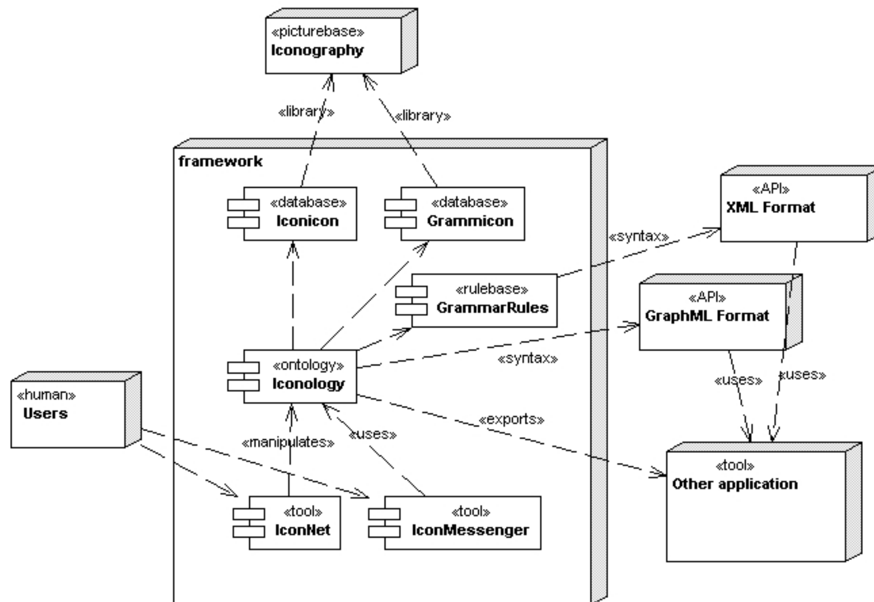


Figure 4.2: System Overview

are listed in subsection 4.1.3.1. The tasks an instant messenger user typically perforsm are described in subsection 4.1.3.2.

4.1.3.1 Visual Linguist

The visual linguist should be able to perform several types of tasks. In the following overview are typical tasks listed (that appear in figure 4.3 too) and are references to sections describing software model functionality given.

- Add normal, relational and grammatical iconic concepts: These are default mutations like described in subsection 3.2.3. In regard to system functionality, this involves the existence of a mutation apparatus. This system functionality is described in section 4.6);

- Alter and delete normal, relational and grammatical iconic concepts: Alteration and deletion are more sophisticated mutations than addition (see subsection 3.2.3.2). Also checking is part of the performing mutations system functionality (see section 4.6);

- Add and change the description and type of an iconic concept: This involves besides a mutation, a dialog to enter information. Dialogs are provided by a panel factory, a class that produces panels on demand (see subsection 4.3.1);

- Assign a new or other picture to an iconic concept: This is like changing the description (see subsection 4.3.1);

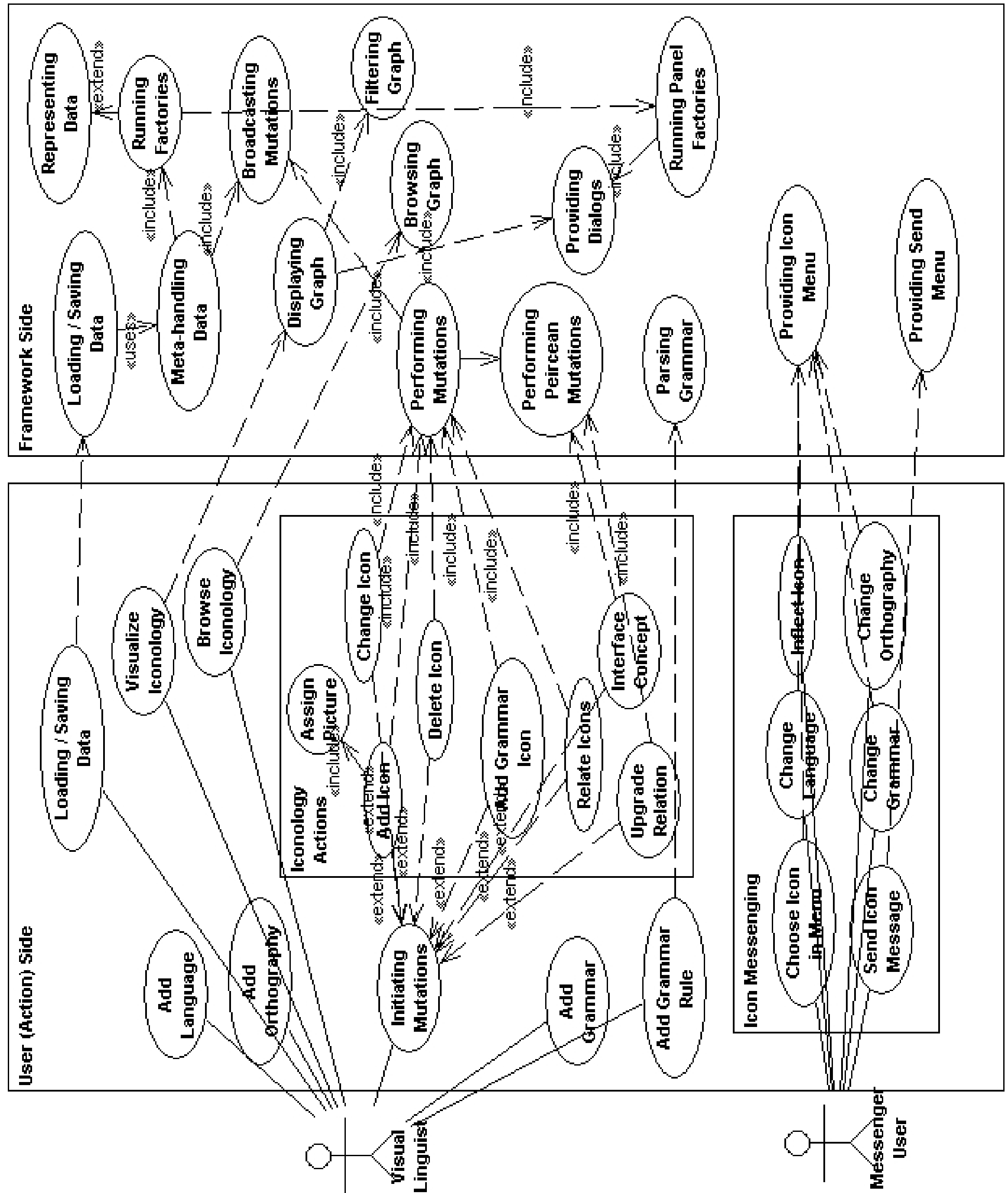


Figure 4.3: From User Actions to System Functions

VilAug IconMessenger IconNet

- Restrict and refine iconic concepts: Restriction and refinement occurs by adding concepts or changing descriptions (see those tasks);
- Relate two iconic concepts (involving addition of a relational concept and indicating its parameters): This involves dragging the mouse from one vertex to another (and becomes implementation specific, see 6.3.1);
- Upgrade a relation (involving change of its type to a normal iconic concept and addition of relational concepts). This is more sophisticated than each of the mutations discussed until now. It involves mouse dragging actions combined with adding and altering multiple vertices and edges in the ontology (a special package is provided in subsection 4.11);
- Interface a concept (involving changing its type and addition of multiple concepts, see also subsection 4.11);
- Add a new language, and a new grammar.

These tasks the visual linguist may perform correlate to specific mutation functionality in the framework. Mutations that can be designed in the software model (see section 4.6). However, many classes are also shown in figure 4.3 that are abstract or general classes that are not one upon one related to actual user actions. The already mentioned panel factory is a good example. This is an abstract class that is able to provide different types of panels and dialogs. These can be used subsequently in the interaction during a mutation event for example.

These abstract classes are used by the above mentioned classes. To this belongs a registry that stores references to important objects (subsection 4.3.3), different types of data representations obeying for example the principle of "verbocentrism" from assumption 14 (section 4.4), graph visualization (subsection 4.7), internal browsing and searching (subsection 4.8) and filtering (subsection 4.9).

4.1.3.2 Instant Messenger User

The tasks of above are not the only set of tasks the frameworks provides. The visual languages defined by the linguists can be used by the instant messenger user. The messenger user should be able to:

- Choose an icon from a hierarchy of icons or icon menu;
- Concatenate icons to strings of icons;
- Broadcast strings of icons, iconic messages, to other users;
- Switch the visual language to another;
- Switch the grammar type to another type;
- Switch the orthography to another;
- Define the way the hierarchy is browsed (grammatical or ontological items as unique beginners of the menu);

- Access in some way the description of an icon (for example by a tooltip);
- Alter an iconic concept in an adjective manner (by altering features like colour).

These requirements speak for themselves. There are not many additional abstract classes behind them, except for some communication commands.

4.2 Software Model

The software will contain two applications that can run independently from each other. The first application is the IconMessenger. The IconMessenger is an instant messenger and can be used to compose sentences of icons and send them to other persons that have the same (or a similar) application. The persons that would enjoy such an application is as diverse as the people that use instant messaging. They are novices in regard to visuolinguistics. The second application is IconNet. This is a graphical user interface for the iconology. People can create and alter visual languages by dragging vertices and edges with the mouse cursor. The persons that would enjoy this application are visual linguists. They can be considered experts in visuolinguistics, or domain experts in a particular field that they want to "iconize" (verbalize in icons). The results of the work of the experts in IconNet will be the input of the IconMessenger.

4.2.1 Introduction

The ontology is built upon the parable of the graph. A graph that contains vertices and edges, and a lot of graph operations are necessary. This graph has also to be visualized and manipulated. To build a model that also embeds all this functionality, would be a pity. It is also possible to connect the model to an existing graph handling package. The graphical package that is chosen for its extensiveness is the Java Universal Network/Graph Framework, from now on abbreviated to JUNG or JUNG package. By someone labelled as very comprehensive: "everything but the kitchen sink!". However, for practical purposes the package is *extended* in this project with a lot of add-ons.

A separation between the part of the model that builds further upon JUNG and the part of the model that is particularly meant for the domain application involving manipulation of icons is maintained. It is therefore possible to publish the first under the BSD License (just as JUNG) and the latter under the *GNU General Public License* (a CopyLeft license). A brief summary of these two main packages in the model will follow in the next subsection.

In the model this syntax will be used for a CertainClass, and a certainFunction will start with a lower-case letter.

4.2.2 IconMessenger Functionality

The functionality that is part of the IconMessenger is like that embedded in a normal instant messenger. Most instant messengers provide emoticons (symbols expressing in particular certain emotions). The IconMessenger uses however only icons for its communication. Although, this may be seen as preliminary, a screenshot of the implemented IconMessenger can be seen in figure 4.4.

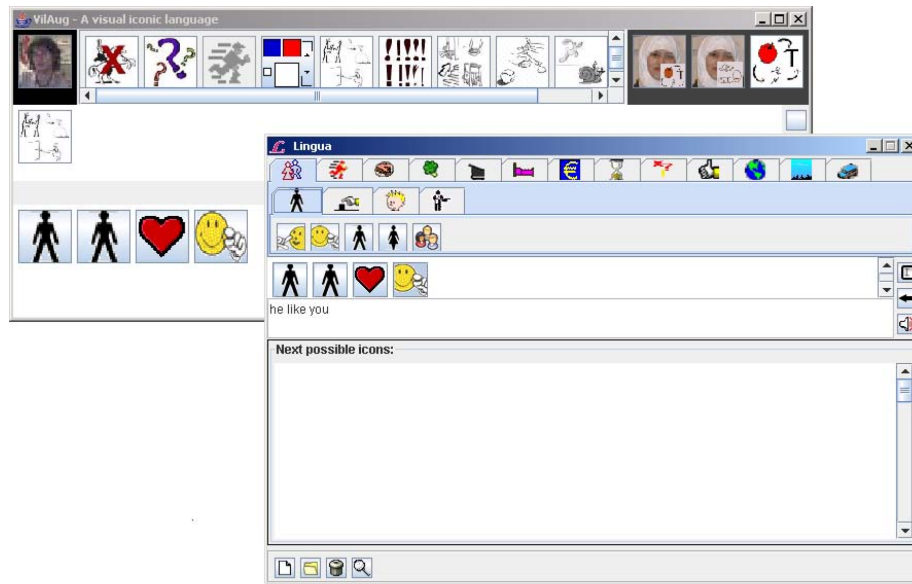


Figure 4.4: Lingua & IconMessenger

The application Lingua as well as the application IconMessenger enable the user to communicate with another user in the form of icon messages. The functionality should be like listed in subsection 4.1.3.2. The tool should also provide a menu, certain menu options, send message buttons and display panels in which a message is constructed and received messages are displayed.

4.2.3 IconNet Functionality

The IconNet tool is meant for the visual linguist. It would be great if the user would be able to manipulate the ontology (according to subsection 4.1.3.1) in a graphical way. Visualizations of ontologies do exist. See figure 4.5 where WordNet is visualized twice.

See also figure 4.6 where a similar application visualizes conceptual graphs (and enables certain manipulations).

The idea behind IconNet is the same. The words should be thought of as icons in that case. In that way a visualization where a lot of icons are related by edges will result.

4.2.4 VilAug & JUNG Extension Submodels

The VilAug submodel contains a main object, the VilAugMain class. The initialization of the application takes place overhere, and the class contains references to all kind of important modules. The software for the IconMessenger and IconNet will not be separated. These applications can therefore be published in two ways. By adding a starting parameter or a starting dialog. Or by adapting this main class so that only the functionality for IconMessenger or IconNet can be evoked. IconMessenger and IconNet should run as applications

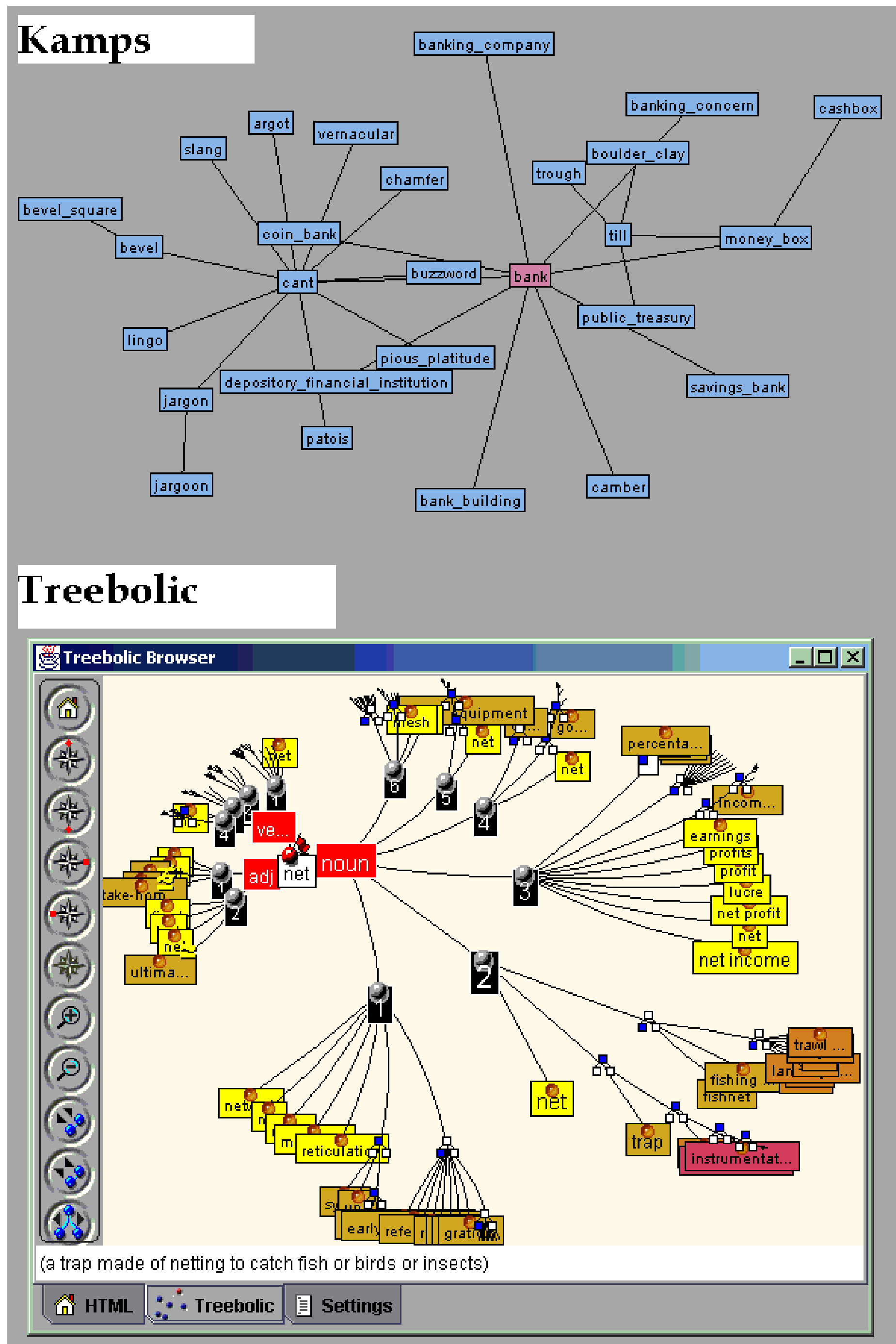


Figure 4.5: Visualizations WordNet

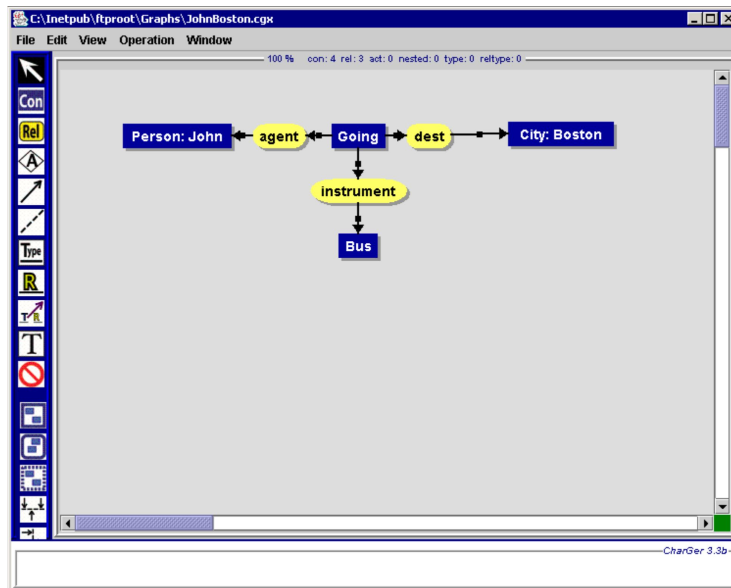


Figure 4.6: Visualization Conceptual Graphs

on the computers of the users. An applet that runs in a browser can serve demo purposes.

The VilAugMain class contains references to modules. Each of these module is an interface and formulates a contract. There are three important modules. The IconModule, the GrammarModule and the IOManager. The IconModule works like a buffer between the parts of the application that do not know anything about the fact that the icons are tied to vertices. This module is particular appropriate for IconMessenger functionality, that does not necessarily know much about the underlying graph structure. It provides functions like getMenuItems, getGrammarIcon and getIconChildren. The GrammarModule contains all functionality needed to apply a grammar. It contains a function that selects from a given set which icons are grammatical correct. Last, but not least the IOManager. (A "Manager" is a class, a "Module" an interface.) The IOManager does provide access to the files and icons on the web or on the local disk. It contains functions like getOntology, getGrammar and getImage.

Another distinction of the functionality of the framework is already more specific. The basic elements of the main packages of the framework are different for each package. A package that has to browse or filter the graph needs vertices and edges. A package that visualize the menu or access the pictures on the disk needs icons. A package that reads the datafiles needs a processing unit. A package that parses the grammar needs a parser. So let us give a summary of the needed functionality:

- JUNG extension package: extensions of graph element classes, adding controller, etcetera;
  - icons: adds icons to vertices, to the graph, adapts visualization like layout, adapts interaction in mousebehaviour, adds mutations apt for

icons;

– graphml: defines default GraphML elements, creates processors for the elements;
– utils: convenient graph, string and set handling routines, basic interfaces;
– mutations: defines mutations upon the graph, defines mutation modes (with sets of mutations), defines mutators (that execute the mutations);
– elements: adds a factory that creates the right type of graph elements (vertices, edges), a checker that checks for acyclicity for example and a dialog that asks for confirmation, additional data or offers information;
– actions: defines a set of action items that can be put in menus etcetera and are tied to defined mutations;
– controller: adds a controller to complete the Model-View-Controller [MVC] design, a registry stores all kind of references to factories, dialogs, etcetera;
– predicates: establishes ways to filter or exclude vertices and edges upon type, parents;
– dag: adds DAG restrictions and defines methods to browse the graph.

• VilAug package: functionality like grammar parsing, icon properties, visualization components and dialogs;

  – grammar: defines grammar constituents (like rule, rule element) and stores the grammar parser;
  – icon: defines icon properties, defines functions that use browsing functionality (checking if an icon is a terminal for example);
  – io: defines search paths, finds files and icons;
  – exceptions: a collection of exceptions that can be raised;
  – components: dialogs and panels for login, logout, icon input, conformations, the icon menu;
  – peirce: mutating functionality that is in the realm of peircean relations (last two graph operations from subsection 3.2.3).

Some files from the JUNG package are also a little bit adapted. These changes are stored in a JUNG Refactoring package, but they are not important for our software model. A graphical overview of the listed packages is given in figure 4.7.

From now on are package names described with lower-case descriptions. Subpackages are separated by their superpackages by a dot. The graphics are created with ModelMaker, that ships with the Delphi suite. However, the UML descriptions are - of course - language independent. The object-based separation of above fits implementation purposes, but let us give a look at the functionality to be modelled. The data has to be represented, the grammar parsed, data loaded and saved, the graph browsed and filtered. These are the topics of the following sections.

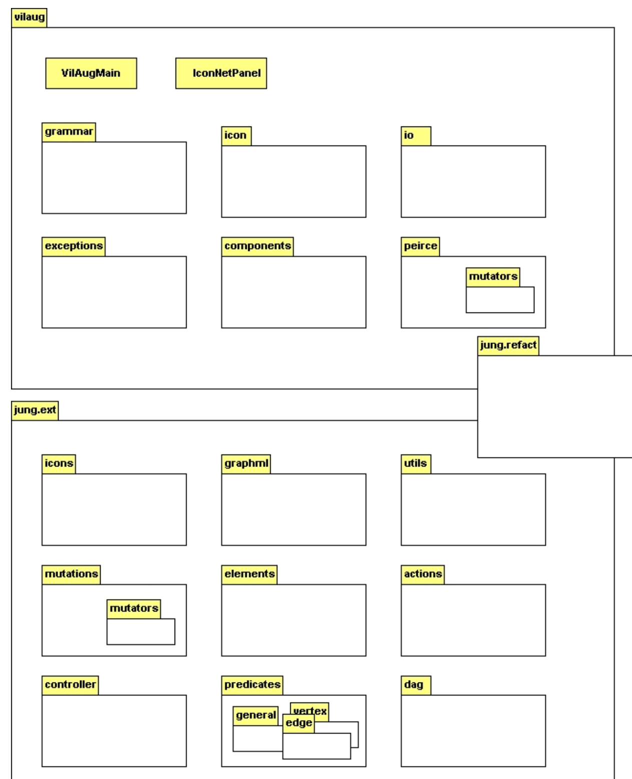


Figure 4.7: VilAug Package Overview

4.3 Meta-Handling Data

Before we start to model different types of functionality, we will describe some general principles and classes. The components that are the topic of this discussion are very abstract, but appear in many packages. They are for example inspired by Design Patterns as defined by the Gang of Four [40] or by other design standards in the programming community. Familiarity with a specific language is not required for this chapter, but familiary with design patterns aids understanding. One method that made it easy to model certain behaviour is to use factories and abstract factories. Factories provide just as in real life certain products, for example a kind of vertices. It is not necessary to define what kind of vertex is used in classes that get their vertices from a factory. And to provide another type of vertex throughout the entire application has only the factory to be altered. Another approach uses a separation between the model, view and controller in an application. The model contains all kind of data representations and data access methods, the view, viewer or viewport contains the classes that visualize things to the user, the controller translates the wishes of the user to changes in the data.

4.3.1 Factories

The use of factories falls under the nomer of *meta-handling* in this report. Meta-handling is provisioned in two parts of the software model. Firstly, meta-functionality aids the creation of graph elements. With graph elements vertices and edges are meant. Secondly, meta-functionality will also be used in handling user actions. And thirdly, it will be used in a certain way in reading the data files. Meta-handling in regard to graph elements is discussed in this section (4.3) about meta-handling data. In regard to user actions it is handled in section 4.6 and in regard to data input in section 4.5.

Meta-handling in terms of a factory is envisioned in regard to the production of graph elements. The vertices can be extended with icons and know a bit more about their parents for example, the edges can be made directed. Changing the products of the factory, changes them throughout the whole application in one big step. The factory that contains this functionality is called the ElementFactory. The package that contains this factory is visualized in picture 4.8.

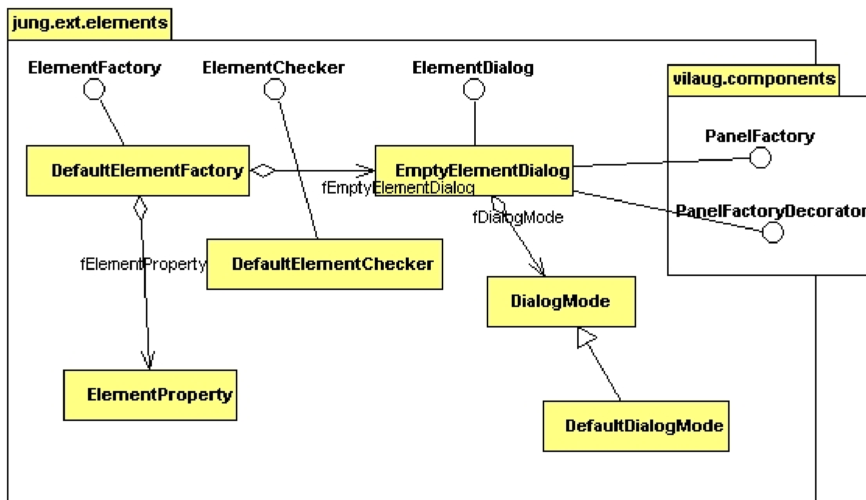


Figure 4.8: Element (Factory) Package

Besides the factory itself, there are also some helper interfaces and classes defined in this package. The ElementChecker checks upon constraints like the direction of the edge in regard to the directed acyclic graph. It evokes an error when a cycle arises. The ElementDialog can ask for information regarding element creation in the factory, or asks for confirmation of certain events, or displays additional information or offers choices. It does make use of another factory that is not mentioned until now, the PanelFactory. This factory provides all kind of panels with labels, editboxes and other stuff. It is also used for logging in and logging out for example.

The ElementDialog is empty by default. Even no confirmation dialogs are shown. To display a dialog this class has to be extended. The DialogMode defines modes for the ElementDialog. These modes vary from “login”, to “con-

firmation", "input" and "error". The layout of the dialog can be altered respectively. The ElementProperty class is needed to add additional information to the elements that come from the ElementFactory. This property can contain information about the edge being directed or undirected for example.

4.3.2 Model-View-Controller

The model-view-controller [MVC] paradigm is applied by defining a VisualizationModel, a VisualizationViewer and a VisualizationController. The distinction between the latter two does not exist as such in the underlying (graphical) JUNG package. The VisualizationModel does store a reference to the layout. The layout on its turn stores the graph. The model contains also methods to set visibility predicates that decide what graph elements from the graph have to be shown on the screen. Finally contains the model methods to add or change data. This involves not only changing the graph itself, but also indicating the layout of the change. The VisualizationViewer refers to a certain model. In this way it knows what graph elements to visualize. It knows how this visualization has to be done. The shape, color, etcetera of vertices and edges can be defined, labels attached, and other rendering performed. The VisualizationController refers to a certain viewer. The controller contains references to classes that want to know about user events that involve addition or alteration of data (see section 4.6). An overview of this design is given in figure 4.9.

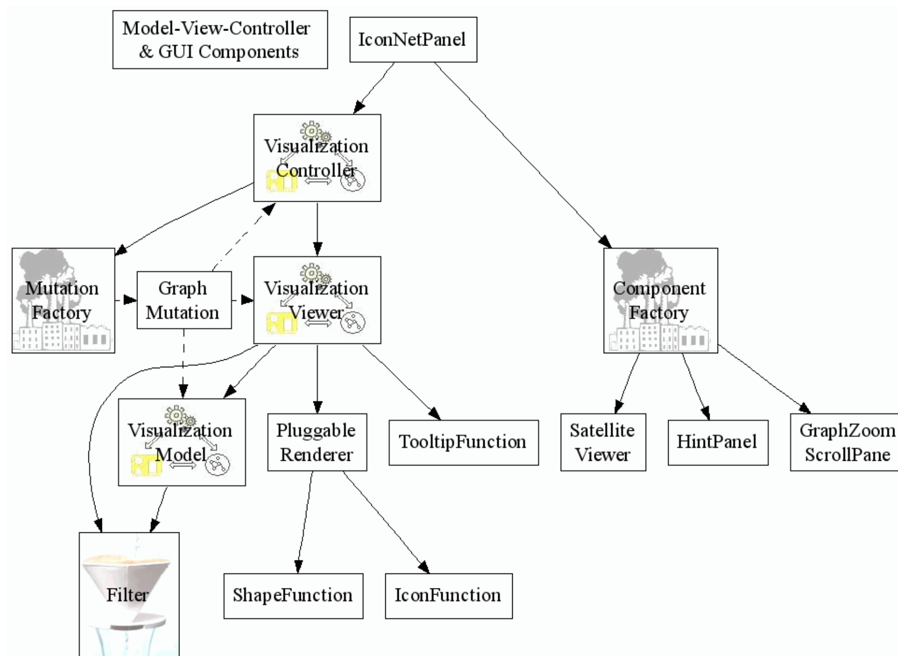


Figure 4.9: Model-View-Controller

The model, viewer and controller are decorated with member classes, like explained. The controller for example contains a certain viewer. This is done in a particular way, that makes it possible to use a kind of registry. This is the

topic of the next subsection.

4.3.3 Registry

A top-level registry, labelled the VisualizationRegistry, provides certain plugin functionality for (especially) classes used for visualization. It knows a range of reusable components, like the MutationFactory, ElementFactory, ElementChecker, VisualizationController, VisualizationViewer, VisualizationModel, Graph, PanelFactory and a few others. It does not know which classes are decorated with the mentioned components. This is not necessary, because all classes implement certain decoration interfaces. When for example a class needs a MutationFactory, it implements the MutationFactoryDecorator. It can subsequently be registered anonymously and the VisualizationRegistry decorates it properly by setting the MutationFactory. In the future only the VisualizationRegistry will be updated to change the MutationFactory through the entire application.

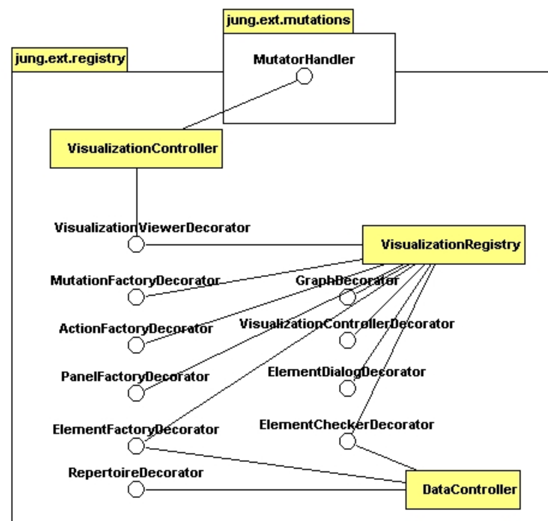


Figure 4.10: Registry Package

The registry assumes that a class like the MutationFactory is unique in the application. All the listeners will receive the updated or other MutationFactory when it changes. If a certain class uses an own kind of MutationFactory, it should - of course - not register itself at the registry. Like the factories and the distinction between model, viewer and controller, is the registry crucial in regard to scalability of the framework.

Now we know some general classes that play a role in the handling of data, it is time to model the data itself. This will be done in the following section.

4.4 Representing Data

The data from the iconology is represented by a graph with edges and vertices. The graphical library JUNG does of course already contain graph, vertex and

edge objects. However, the IconGraph and IconVertex classes have to be modelled still. A distinction is maintained between the classes where the *iconic nature* is emphasized and where the *graph representation* is emphasized. The latter classes are part of the JUNG Extension package, the former are part from the VilAug package. In subsection 4.4.1 will the graph representation part be modelled, in subsection 4.4.2 the iconic representation package and although of secondary importance in subsection 4.4.3 the verbal representation part.

4.4.1 Graph Representation

The main data components in the JUNG Extension package are visualized in figure 4.11. The IconGraph class adds labelling to the vertices and edges it contains. It also implements a convertor that enables it to move to and from the vertex presentation to the icon presentation of an IconVertex. An IconVertex is a combined icon and vertex in one, with the emphasize on the latter. It does have its twin in the VilAug package (that is called VertexIcon). These Janus-faced presentation enables the use of an appropriate representation for different types of problems. In browsing the graph (see section 4.8) the vertex presentation can be used. When visualizing an icon upon a button the icon presentation can be used.

The convenience of the IconVertex lays in the fact that it builds further upon the FamilyVertex. The latter class is part of the JUNG Extension package that particularly fits the realm of directed acylic graphs. The IconVertex has also a type parameter that denotes its resulting menu visibility. There is a type for relation-like concepts, for visible concepts and for interfaces. Moreover, the IconVertex is of course combined with an icon to denote its concept, but it also contains a picture denoting its type. And it has a string as tooltip, that can be used when the mouse hovers over the icon.

The iconology does contain only directed edges, because relations are visualized as nodes (see previous chapters). The edges are therefore very simple and need no additional modelling.

In figure 4.11 is also an IconGraphMLElement shown. This comes from the fact, that the graph should be populated with data from the module that reads the GraphML files (see section 4.5 about loading and saving). The class that defines the details of a component of this GraphML representation is IconGraphMLElement. It scans the GraphML file upon graph, vertex and edge elements and creates these objects in the application. Therefore it stores references to an element factory. The specific IconGraphMLElement enables specific behaviour like taking the union of several graphs in the input files. More can be found in the already referenced section.

This subsection described only the top-level graph representation. More specific details, for example about the FamilyVertex and its functionality, can be found in subsequent sections. The FamilyVertex is further described in section 4.8 about browsing the graph.

4.4.2 Iconic Representation

The VertexIcon is a graphical class that can be painted upon buttons. It is part of the VilAug package (see figure 4.12). It contains methods to retrieve the corresponding IconVertex, to set its tooltip description, and to retrieve the

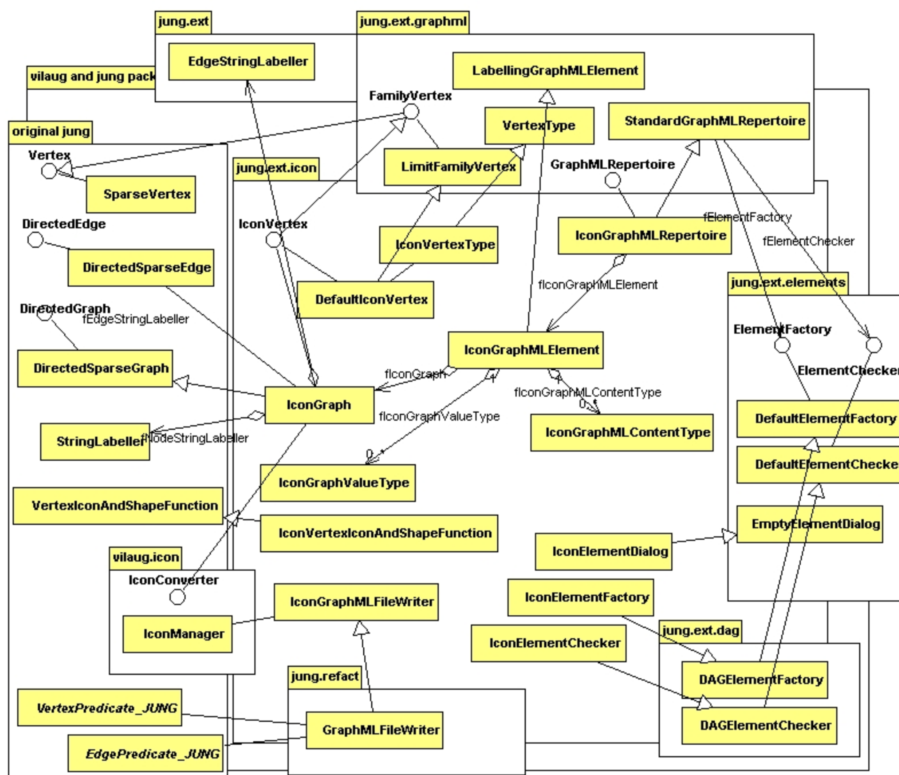


Figure 4.11: Icon-based Extension of JUNG

filename. The filename has to be given on instantiation. The IOManager will be asked for an icon given that filename and return an icon or dummy icon. It is never the case that nothing will be painted on the screen at all.

By the way, both presentations, the VertexIcon as well as the IconVertex are interfaces. The details of the latter will spring from browsing requirements. Implementation details of the former originate from other requirements. The class should be able to handle .gif and .jpg picture formats, it should use *lazy image loading*, what means that the picture is only loaded from the file when it has to visualized, and it should use *image caching*, what means that the picture is stored locally once it has been downloaded from somewhere else. These characteristics has the DefaultIcon class.

The functionality of the DefaultIcon class is also captured in the underlying DefaultImageIcon class. This class may be platform specific, but has to fullfill several additional requirements. It contains a method that loads an image from the IOManager. This should be tried several times and on continous failure a default dummy icon should be loaded. It also contains methods to scale the icon. The icons can therefore be stored in several directories in prefixed sizes (like 32x32 or 64x64). Above that it knows how to add multiple images in row to the same icon. In this way image combinations can be made for one and the same icon.

When all functionality is needed except for the reference to the vertex rep-

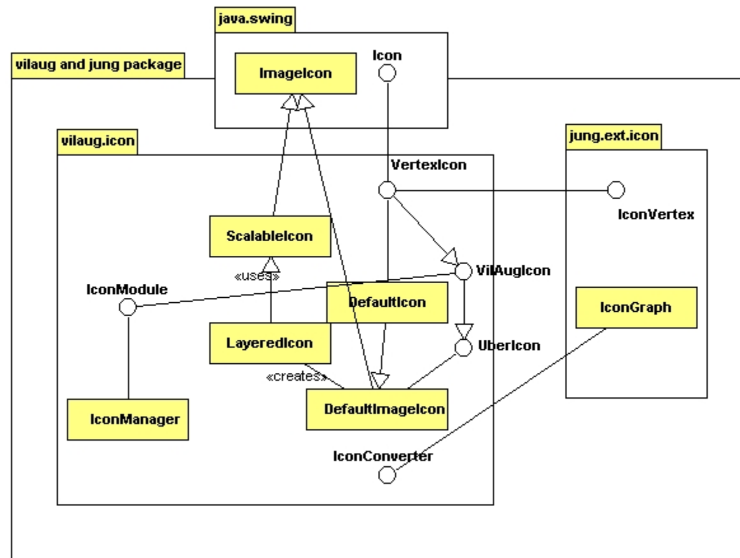


Figure 4.12: Icon-based VilAug Package

resentation, the interface VilAugIcon can be used. The use of this class is especially appropriate for general classes. It encourages entire other data representations of icons, that nothing have to do with graphs.

4.4.3 Verbal Representation

A brief note about an also existing verbal representation. Verbocentrism (see assumption 14) is a resistant danger in the software design process. There are however situations where verbal labels are used for identification of iconic concepts instead of icons themselves. This in particular in regard to grammar (see section 4.10). The software developer should refrain from the use of this and therefore a peculiar interface has been created, namely VerboCentro. This interfaces does only have a function that returns a label. There are two types envisioned, a GrammarVerboCentro class and an ApplicationVerboCentro class. The former can be used to translate from the grammatical descriptions in the grammar rules files to iconic concepts. The latter can be used for icons that are also used in the application. Like an icon for “help”, “okay” or “grammar”. This VerboCentro class can also be found in figure 4.12.

Now we know the tree types in which data can be represented, let us look how the data is loaded and saved.

4.5 Loading & Saving Data

The data is stored used a standard format for graph representation, namely GraphML. To be able to use the same representation with other applications a GraphML API has to be defined. To populate the graph from these files the

GraphML has to be parsed. This is described in subsection 4.5.2 and a UML diagram is given. The section ends with the way file modularity is envisioned.

4.5.1 GraphML API

Let us first describe the API (Application Programming Interface) that is needed when other applications want to use the same ontological data. The data that describes the graph, the vertices and edges and additional information like a tooltip. The GraphML file format is easy-to-use and able to describe the structure of a graph and its elements. In the GraphML primer is the syntax described, see "algorithm" 1.

Algorithm 1 GraphML File Format Syntax

```
<?xml version="1.0" encoding="UTF-8"?>
  <graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
    http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">

  <key id="filename" for="node" attr.name="filename" attr.type="string"/>
  <key id="tooltip" for="node" attr.name="tooltip" attr.type="string"/>
  <key id="visibility" for="node" attr.name="visibility" attr.type="string">
    <default>normal</default>
  </key>

  <graph id="IconGraph" edgedefault="directed">
    <node id="900047225">
      <data key="filename">vilaug.jpg</data>
      <data key="tooltip">the framework VilAug</data>
    </node>
    <node id="1084049630">
      <data key="filename">grammars.gif</data>
      <data key="tooltip">collection or set of (visual) grammars</data>
    </node>
    <edge id="-284923241" source="900047225" target="1084049630"/>
  </graph>

</graphml>
```

The GraphML file starts with references to verify its contents. Then a <key> element appears. These are customizable data elements, called GraphML attributes - that can be attached to vertices or edges. For our purposes attributes with the name "filename", "tooltip" and "visibility" are defined for all vertices. Nodes and edges have identifiers. These identifiers are numbers, this decreases the danger of verbocentrism. Manual altering the files is a tedious thing. To know to what vertices an edge connects, one has to search for the corresponding identifiers through this entire file (or even other files). The GraphML files are however meant for machine-machine interaction, not for machine-human interaction. A human could very likely corrupt files or violate acyclicity or reusability constraints when manually altering these files. This danger is reduced by using

an intermediating machine. The filename does not contain path infomation, but can have an extension. The tooltip gives an approximation of what the icon means in an English sentence. It tries to capture nuances of the iconic concept.

How we get the information from the files into the framework is described in the next subsection.

4.5.2 GraphML Parsing

Classes that very much resemble factories are the GraphMLElement classes. The classes in the GraphML package handle the access of the graph data in the files and should know how to handle the GraphML syntax (as described in algorithm 1). The GraphMLElement interface functions as a kind of representation of the defined <element> objects in the GraphML file and a kind of processing unit at once. In the latter role an implementation of the GraphMLElement interface can contain a labelling unit. Labelling is not needed for graph objects, but it is for vertices and edges. If there are no labels assigned to vertices and edges there is no possibility to check if they are unique for example. There is one function, getCargo that returns an vertex, edge or other element, whatever is appropriate.

An GraphMLElement can be assigned a certain type (GraphMLElementType) that stands for the kind of <element> that has been encountered, like <graph>, <node> or <data>. Subsequently a method setCargo can be called to set the XML-attributes corresponding to that element, like "id=1093284". These methods are called from the parsing object, the GraphMLFileReader. It receives notifications of parser events from the SAX (Simple API for XML) units. The GraphMLFileReader pushes an element (with its attributes) that it encounters upon an ElementStack. It continues to do the same with subelements it encounters. It pops the element back from the stack when it comes at the end of an element in the GraphML file. That is the moment the element is actually created and populated with data.

An object that also resembles a factory and is part of this package is the GraphMLRepertoire interface. It returns a specific type of GraphMLElement instance. It may return an element that knows how to take the union of several graphs in the source files. Or it may return an element that stores references to yet unknown graph elements until they are encountered. Or other behaviour that is more elaborated than only the of another type of vertex or edge.

4.5.3 File Modularity

It is already mentioned and visualized that all kind of icons and types from all kind of domains are stored in one iconology (see subsection 3.2.1.4). To make maintenance effortless, the concepts are stored in several files. With file modularity are all icons that a certain visual linguist defines or adds stored in an additional file named after her name or given the name of the visual language. Also the edges that link the normal and relational concepts are stored in that file. Likewise the edges that link to iconic concepts in the standard library. The standard library assures some consistency and offers a visual language core to start. Dependencies between languages are defined in the top of the custom GraphML files. The grammatical concepts can be stored in another additional

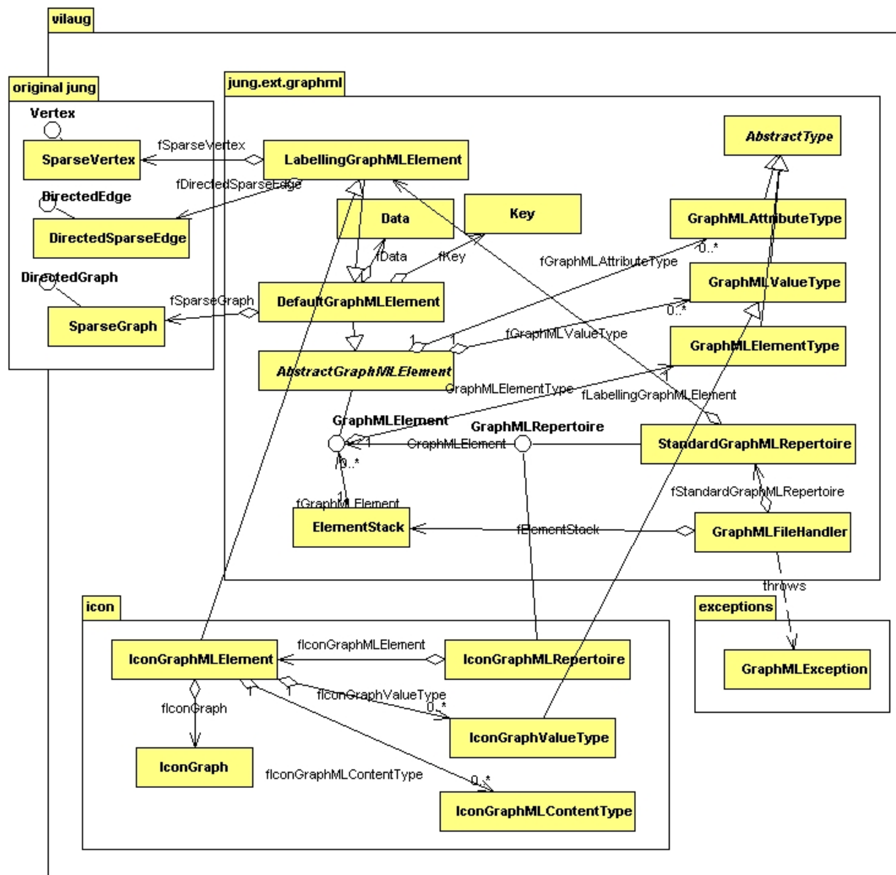


Figure 4.13: GraphML Package

file. And last but not least the grammar rules have to be defined in an additional file.

The order in which the files are loaded is predefined. Edges without nodes cause a failure in loading a subgraph. Duplicate edges and nodes evoke a warning. This modular approach enables iconic linguists to attribute to this project simultaneously. To design a client-server pattern is not difficult either using these different files. The core can be situated on the server and the client applications can download certain visual languages or create a specific language on client machine.

Recommendation 33. *The IconGraphMLHandler should load the files using the dependency lists. It should itterate through the grammar directory, order the files and load them subsequently.*

The next section will describe how the data can be altered by the user when the framework is running.

4.6 Performing Mutations

In subsection 4.1.3 and 3.2.3 are tasks, respectively mutations, defined. With mutations are in general only the user actions considered that alter the data in the graph. This can be addition, deletion or alteration and all kind of combinations of these actions. These mutations and mutation bundles come from the part of the application that receives mouse events. From there the mutations are broadcasted through the entire application to the object that holds the data. This process is not straightforward. The application is designed according the MVC paradigm (see subsection 4.3.2). The mutations have to be send from the controller (in the MVC structure) to the model (in the MVC structure). Hence the need for an elaborated mutation package.

4.6.1 Mutation Types

There are many mutations; very complex mutations involving the addition and alteration of several graph elements can be defined. The mutations are therefore also produced by a factory (like the graph elements themselves in section 4.3). Every class of plugin can get in this very convenient way access to a specific set of mutations from what is called the MutationFactory. The MutationFactory knows how to fabricate GraphMutation objects. A GraphMutation does have a vertex, edge or other graph element as cargo. Besides that it has a certain MutationType, like “create edge”, “remove vertex”, etcetera. To know the type of graph element that has to be created, the MutationFactory is on its turn, connected to an ElementFactory. The MutationFactory also connected to an ElementChecker and ElementDialog to provide checking routines and dialogs. These four abstract classes work together smoothless to initialize a mutation. An overview of the mentioned objects is given in figure 4.14.

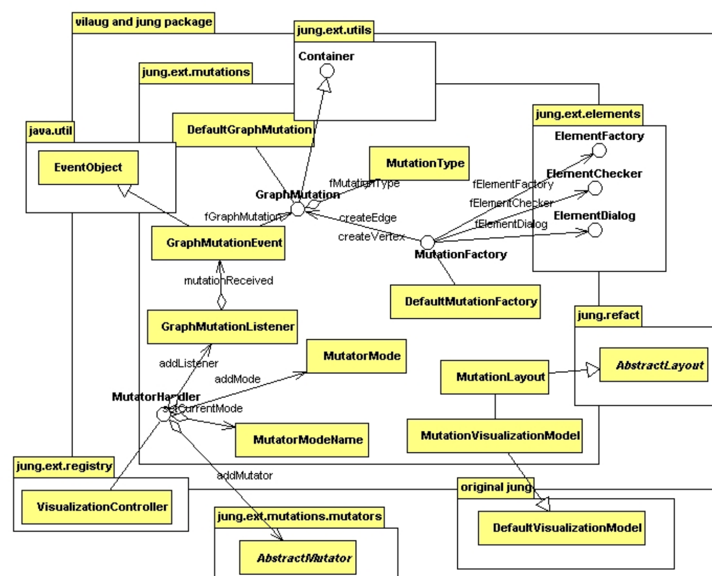


Figure 4.14: Mutation Package

The mutations are now defined and intialized. Broadcasting and actual performing the mutation is the next action. This is the topic of the next subsection.

4.6.2 Mutators

The main object that has to do with broadcasting the mutations is the already mentioned (subsection 4.3.2) VisualizationController. It keeps track of a set of listeners to mutation events that are called mutators. When a mutator hears a mutation, it checks its type and acts correspondingly. The mutator can be added on a solitary basis or using a mode. A certain mode with multiple mutators can be added at once to the VisualizationController. For example it is possible to add a mode with the name “addition only” and a MutationMode object called “creators”. Another mode like “all mutators” can exist out of a collection of other modes. A hierarchy of mutators is created, see figure 4.15.

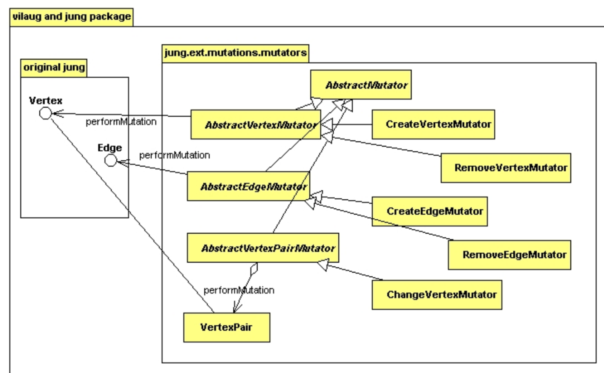


Figure 4.15: Mutator Package

The mutators defined in the framework will later be extended with even more types. The ones over here are the basis of most others, they all inherit from AbstractMutator that is a GraphMutationListener. The AbstractMutator knows a MutationFactory and a VisualizationModel. It receives a GraphMutationEvent that contains a GraphMutation. In the derived AbstractEdgeMutator and AbstractVertexMutator classes, certain additional checking routines and dialog opportunities are provided in the code, and subsequently a performMutation(edge), or performMutation(vertex) method is called. The AbstractVertexPairMutator can be used to carry two vertices (a VertexPair) at once. For example when a vertex has to be changed by another. The CreateVertexMutator, CreateEdgeMutator, RemoveVertexMutator, RemoveEdgeMutator and ChangeVertexMutator classes derive from the mentioned ones.

In the performMutation method of the mutator the VisualizationModel is used to execute a mutation like removeVertex or the MutationFactory is used again to redirect the mutation in the form of several smaller mutation objects. When the VisualizationModel finally receives the appropriate information it transfigures the graph in a corresponding way.

VilAug IconMessenger IconNet

4.7 Displaying the Graph

The working horse in displaying the graph is the VisualizationViewer in the (graphical) JUNG package. Framework specific functionality is limited to the layout of the graph elements. An approach resembling navigation through a tree menu (like in a graphical disk explorer) is designed. The items in such a treeview expand and collapse on doubleclicking. Expansion means that the subitems of a certain superitem are visualized. The space between superitems is a bit enlarged and the subitems are put in between. A graphical equivalent is not much different. The items are in this case not vertically aligned, but positioned in circles around a superitem. Doubleclicking a vertex expands it and shows a circle of subitems. Again doubleclicking makes it collapse.

The layout functionality is written down in the EccentricLayout and the corresponding EccentricData class. The graph representation should account for two types of vertices. Visible concepts and relational or grammatical concepts invisible in the menu. The latter have still to be visible in the graph representation. They are therefore rendered as small red circles. Another characteristic is that relational concepts are on a circle a little closer to their superconcept. The EccentricData class labels this difference in terms of “gender”. The relational concepts are the boys, the others the girls. The position of an arbitrary vertex depends on several factors:

- The amount of parents;
- The last added brother and sister;
- The amount of generations in the longest (visible) lineage.

The parent is needed to know the center of the circle. The index of the next brother or sister has to be known to know the angle. And the amount of generations in the longest lineage is important to adjust scaling (radius of the circle) and use of white space. There can be opted for swapping subitems in such a way that it is less likely that they collide with neighbours. Or the entire graph can be scaled, such that the radius of a circle belonging to a generation is much smaller than its parent generation. Or the distance to the center of the circle can be enlarged when the vertex expands.

4.8 Browsing the Graph

The graph and vertex are already modelled in subsection 4.4.1. Many details are left out, because they are especially tailored at navigating and querying the graph. The graph is a directed acyclic graph (see subsection 3.3.2). Certain operations like browsing upwards or downwards in regard to the directed edges are definitely part of the repertoire. This kind of operations are intuitively grasped if they are formulated in terms of a family analog.

4.8.1 Family Analog

The family analog is used in finding and returning “children”, “parents” and “ancestors” of iconic concepts. It is only used in the context of the strict subsumption relation of subsection 3.1.3.1. The icon higher in the higher is called

the "parent" of the icon that it connects to on a lower hierarchy level. The direction of the edges is from the higher levels towards the lower levels. An icon can point to several parents, and to several children, but it should obey the acyclicity constraints. Functionality that has to do with this lattice structure is stored in a DAG package, see figure 4.16.

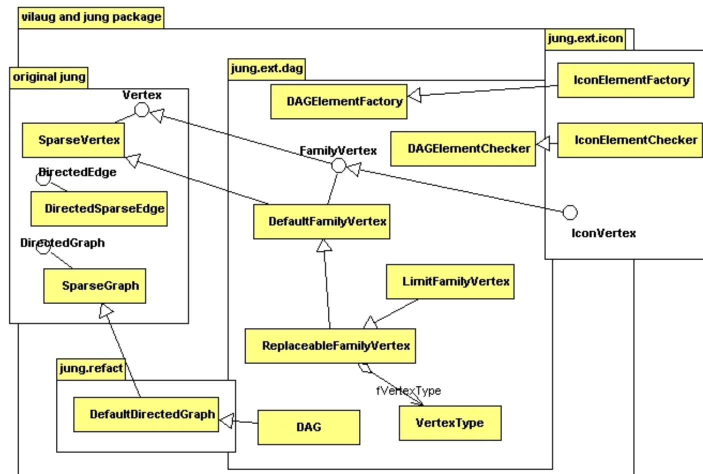


Figure 4.16: DAG Package

The browsing functionality is contained in functions added to the FamilyVertex interface. The main functions that operate on this vertex are getChildren, getParents and hasAncestor. Two flavours of the getChildren methods are envisioned. One without parameters that merely returns the children of the vertex. One with a parent or ancestor as parameter, that returns the children that both vertices have together.

4.8.2 Different Node Types

In figure 4.16 several implementations of the FamilyVertex are shown. In the DefaultFamilyVertex class the getChildren and getParents functions are equal to the already existing getSuccessors and getPredecessors functions from the (graphical) JUNG package. It also contains additional functions like returnParents that returns parents of a vertex from a given set of vertices. Another additional function is hasAncestor like described in the previous subsection. It navigates through the hierarchy towards the top, searching for a certain ancestor.

The ReplaceableFamilyVertex is an implementation that knows how to treat relational nodes and interfaces (see subsection 3.1.3.4 and 3.2.1.4). Although relational concepts are first-class citizens just as other concepts, they are treated differently, because they never show up in the icon menu. They are always invisible. The ReplaceableFamilyVertex class does therefore return for the getParents method all the parents of a certain vertex that are connected to it by a relational node. Or in another point of view, it replaces the invisible nodes by its parents (or by its children) when appropriate.

VilAug IconMessenger IconNet

4.8.3 Time and Speed

In the application certain specific nodes are queried quite often. It is therefore useful to optimize these functions in regard to speed. The hasAncestor and other similar functions should for example be searched from the bottom to the top. Most nodes have only one or two parents, but many children. This manifests itself even by the non-existence of functions like hasDescendent (only its reciprocal hasAncestor exist).

Another way to gain speed is to add restrictions to the search algorithm. A kind of filters are added in the form of particular nodes. So, is it possible to define nodes that cause the search methods to stop browsing the hierarchy in a certain direction. For example, if it is known that PEOPLE can not be the ancestor of SEALION, than can PEOPLE be appended to the set of restricting vertices. These vertex are collected in filters. The implementation that uses these filters is the LimitFamilyVertex. More information about filters can be found in section 4.9.

The searching methods check a predefined filter or predicate. This "browsing predicate" can be adjusted by adding including or excluding material. The method setStrangers does for example limit the search algorithm in the described manner. Searching for kinship will halt, when a "stranger" is reached in the tree. The algorithm will continue with the "aunt" of this strange vertex. In a similar manner can "elderly" added by the method setElderly. These nodes belong to the ancestry of a node, but their kinship is too remote to investigate the tree further overthere. Also here will the search be continued with an aunt. Another option is to define fornicators. Their ancestry is not considered as genuine offspring, but as baseborns. So, all their descendents will be disregarded from the search. Instead of continuing the search with the aunt of a certain vertex, a whole other branch will be taken. So a fornicator is even more limiting than a stranger. These are only a few examples of what kind of filters can be designed to increase the speed of searching.

4.9 Filtering the Graph

The filtering routines are not limited to the realm of navigating the graph, in contrary. Filtering is mainly used in the visualization part of the framework. The layout calculates positions for only a limited set of the total amount (thousands) of vertices available. To decide which vertices should be visualized and which will not be visualized, a filter or predicate is added to the layout. This predicate is part of a predicate package, see figure 4.17.

There are a few basic predicate classes in the main predicate package, but the classes that are most often used are in subpackages. The general subpackage contains the DefaultPredicate class. This is a class that is a particular form of the Predicate as defined in the Apache collection. It contains however much more functionality. Firstly, it is *tailored to graph elements* by the method evaluateElement (by extending ElementPredicate). Secondly, it uses *preconfiguration* of the to be computed elements (by extending PreconfiguredPredicate). This means that the evaluateElement function only needs to check if the given element is part (or is no part) of a precomputed set of elements. This precomputed or preconfigured set has to be calculated before. This is a tiny disadvantage,

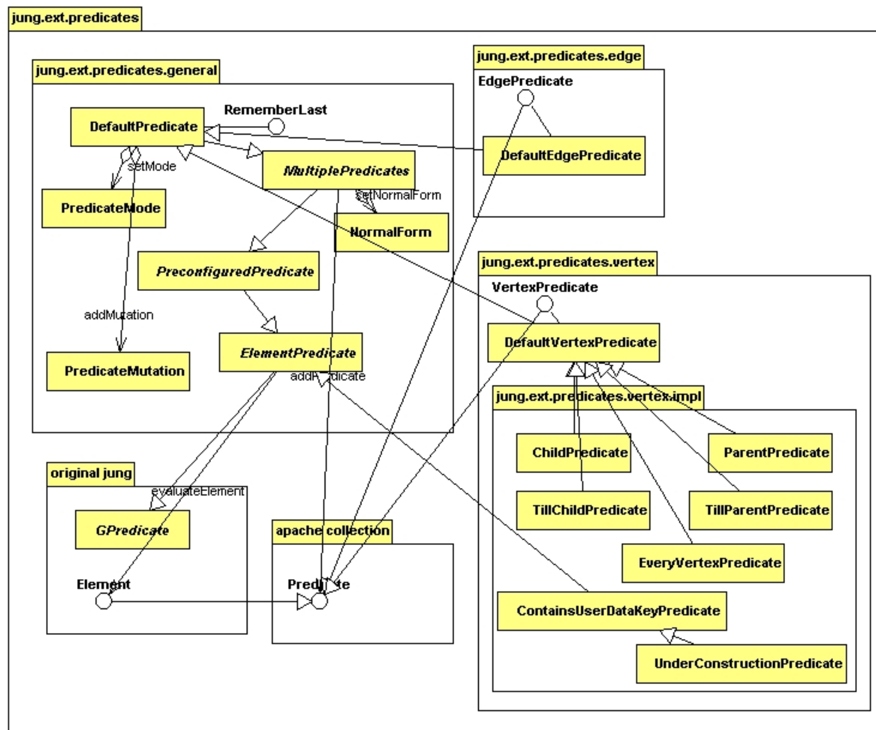


Figure 4.17: Predicate Package

because the application has to know when a change in the result of the predicate is likely. But preconfiguration is at the other hand a big advantage because filtering would otherwise imply continous checking of thousands of graph elements in the visualization routines. In the context of the framework indicating the predicate of a change is possible. The model receives all notifications of mutation events and changes in the graph. The model can therefore propagate this information to the predicate too.

Thirdly, the predicate can contain *subpredicates* (by extending MultiplePredicates). It is possible to add a certain predicate to an existing predicate. The result will depend on the combination of these two predicates and is defined by a NormalForm (which can be conjunctive using an AND-operation or disjunctive using an OR-operation). During preconfiguration not a mere set of elements will be used, but in this case the computed predicate will be pre-evaluated. Thus this is a two-stage process. The predicates are directly combined to one large predicate. And then - when evaluateElement is called and the predicate is not uptodate - this predicate is transformed to a set of (dis)allowed vertices.

Fourthly, the DefaultPredicate does have certain default methods, like useGraph that indicates that it takes elements from the graph itself, instead of a given set of graph elements. It also adds *listening functionality* to subpredicates that are DefaultPredicate instances. Adding the superpredicate as listener to the subpredicate means, that if the application raises the update flag of a certain subpredicate, the superpredicate will automatically update itself too, when time

VilAug IconMessenger IconNet

is near. The DefaultPredicate contains also some *memory*. When the predicate is in "recording mode" the results of the previous and the current preconfiguration are stored. The difference between these two sets can be accessed by the method getOutputChange. It is for example easy to have one updateAdditions method in the layout. It only knows that there are some more vertices to be visualized and to retrieve them by getOutputChange. It does not have to know how exactly the predicate has changed.

4.10 Parsing Grammar

In parsing the grammar the type of the grammar is important. It directly imposes restrictions upon the type of the grammar parser. The model of the grammar parser is given in subsection 4.10.1. The GraphML API in regard to the storage of data is already described in section 4.5. However there is also the representation of grammatical rules. Also this kind of information should be shareable between applications. The Grammar Rules API will therefore be described in subsection 4.10.2.

4.10.1 Earley Parser

In subsection 3.1.4 on page 56 and 3.3.3 on page 68 some theoretical background and requirements are given for parsing the grammar of a visual language. The actual parser that is used in the software model is a chart parser. More specific, it is an Earley parser that is from a mathematical viewpoint a push-down automaton [PDA] and from a practical viewpoint an implemention of an $O(n^3)$ algorithm for parsing. The Earley parser makes use of three function blocks or stages. These functions can be distinguished in an object-oriented style. The stages are called: *prediction*, *scanning* and *completion*. The general layout of such a parser is given by Russell and Norvig [41].

The Predictor is a system element parsing *top-down*. It tries to specialize or expand every sentence at the current position (modelled by the "Dot"). It specializes by replacing an element by all the production rules that lead to that element. As a result the rules grow bigger when the Predictor passes. In this software model the Predictor has also another function. It not only lists all possible continuations of a certain sequence. It does also return a list of the next possible symbols. It offers a set of expected symbols. Another system element is the Scanner. The Scanner starts when prediction (and completion) is finished. It records a new symbol and updates the Dot. Now the Completer starts to work. The Completer goes *bottom-up* and tries to apply the production rules on the elements before the current position. It generalizes by replacing phrases by elements.

Also this grammar can be described in the context of a graph. The described system elements will alternately add edges and vertices. With each icon that is parsed a vertex is added to the grammar graph. Yet unfinished production rules are added to the edges. This does have side-effect that even the parsing algorithm can be investigated in a visual manner by the visual linguist.

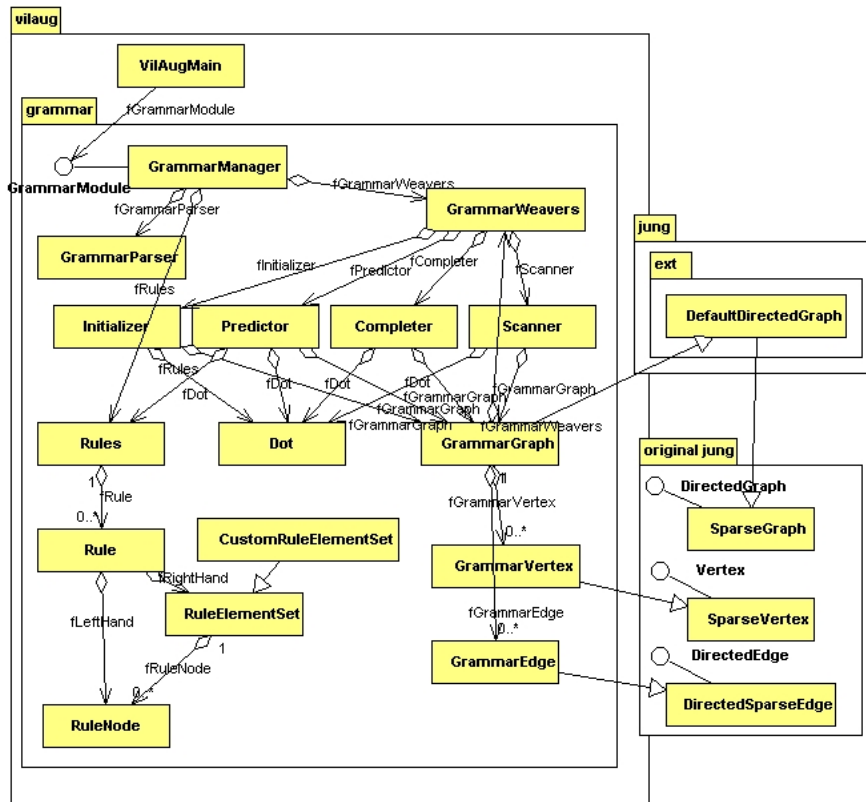


Figure 4.18: Grammar Module

4.10.2 Grammar Rules API

The syntax of the grammatical rules is like BNF grammar with an XML flavour. The Grammar Rules API is as in .

When this syntax is compared with the GraphML syntax it is clear that some kind of translation to the identifiers in the GraphML file is needed. The syntax of these rules has to be friendly towards visual linguists. The linguist can use grammar tooltips instead of identifiers. To reduce the risk of verbocentrism the following approch is recommended:

Recommendation 34. *Implement a GUI that describes grammar rules with icons instead of verbal phrases. In that case a visual linguist will never see an XML grammar file and identifiers can be used without concerns about usability.*

Such a graphical user interface is not added to this design in this stage. It would have involved icons to denote optionality and phrases of other icons.

4.11 Peircean Functionality

An additional package of mutations and mutators contains is aimed at relational concepts in a Peircean flavour. The relational concepts contain a representamen,

referent and interpretant. The user actions correspond to specific changes in this structure. As described in subsection 4.1.3 this entails mutations like "create relation", "upgrade relation" and (although having less to do with Peirce) "interface concept". These mutations and mutators are collected in a Peirce package, see figure 4.19. This package is specific to the iconic domain and therefore part of the VilAug superpackage.

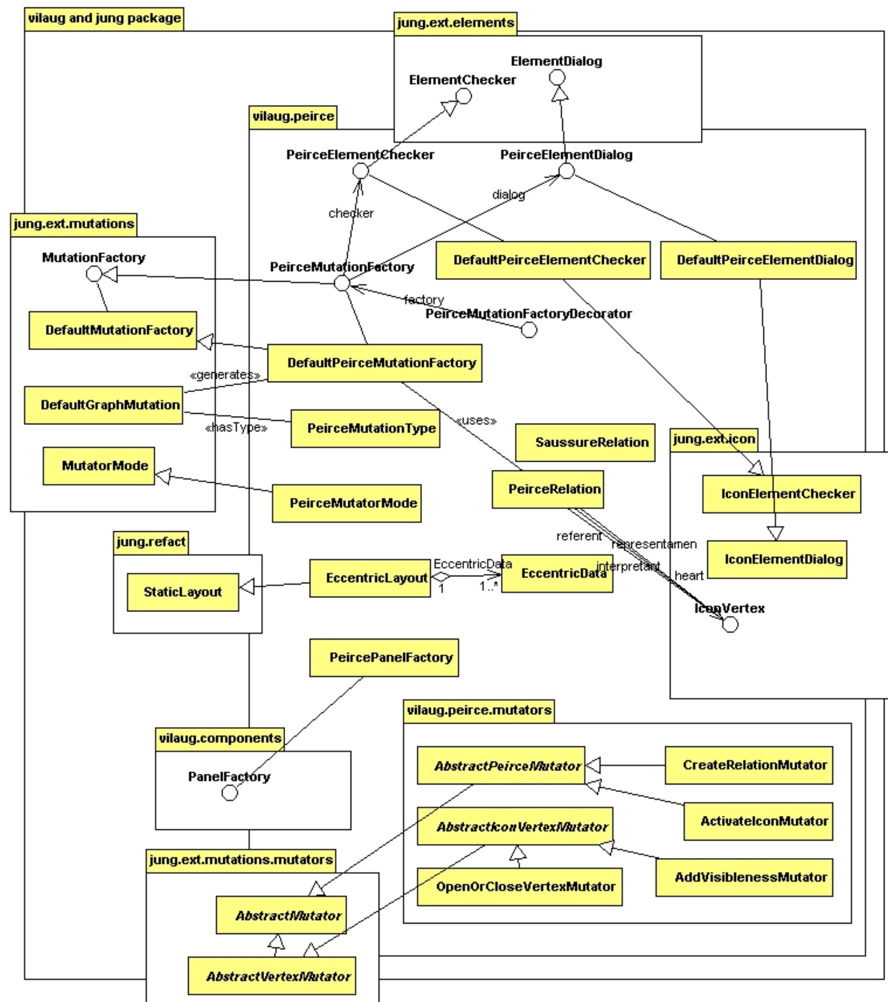


Figure 4.19: Peirce Package

This package does not only contain an adjusted set of mutations and mutators, with a PeirceMutationFactory, PeirceMutationType, PeirceRelation and PeircePanelFactory. Particular sets of mutations are collected together in PeirceMutatorModes and added to the VisualizationController.

The Peirce package contains also the EccentricLayout described in section 4.7. And to check the relations and display a dialog regarding relation data a PeirceElementChecker and PeirceElementDialog are incorporated. The PeirceVisualizationModel contains functions to expand and collapse vertices in the

way described in section 4.7. It also contains functions inspired by biological process, namely applyBiproduction and applyMeiosis. The former generates a vertex from two other vertices, the latter derives a vertex from one other vertex.

The EditingPeirceGraphMousePlugin contains instead of methods like mousePressed(vertex) equivalent methods in the domain of Peircean relations. Methods like interpretantPressed(interpretant), referenced(representamen, referent), referencing(representamen), etcetera. The behaviour can easily be adjusted by implementing these functions.

With this specific part of the software design chapter especially aimed at the iconology ends this part of the report. The following part will elaborate upon issues that are implementation specific, language dependent, or in another way inappropriate for this software design chapter.

VilAug IconMessenger IconNet

VilAug IconMessenger IconNet

Part III

Implementation

Summary Part III - Implementation This part of the report contains two chapters about the implementation of the framework. Chapter 5 on page 107 describes the implementation in the form of use cases for the programmer. These are used to explain implementation instead of a detailed description of each of the 250 created classes. How to browse the iconology (subsection 5.3.1), how to implement a (new type of) user action (subsection 5.3.2) and how to layout (subsection 5.3.4) are use cases that are discussed.

Chapter 6 on page 125 gives a graphical tour of the implemented framework. The graphical user interfaces of the IconMessenger and IconNet tools are shown. The graphical tour does also use two use cases. The case of adding an iconic concept (subsection 6.3.1) and the case of upgrading a relation (in subsection 6.3.2).

This part about the implementation of the framework will be followed by part IV on page 135 that evaluates the model, its implementation and the implemented tools.

VilAug
IconMessenger IconNet

Chapter 5

VilAug Implementation

This chapter describes the implementation of the VilAug framework. After the introduction follows section 5.2 with an overview of the packages in the framework. Section 5.3 describes the implementation in a way that enables to explain most important implementation facets, namely by recognizing the ways a developer may extend the functionality of the framework.

5.1 Introduction

This chapter does describe the implementation of the VilAug framework. The UML descriptions with additional explanations are given in the previous chapter. What follows are code snapshots from certain specific packages and classes part of the entire framework containing over 250 files. More exact and detailed information can be found in de *javadoc* commentary appended to each class and method in the source code. Thousands of explanatory comments can be found over there. In appendix E an overview of all implemented classes can be found.

The framework is divided in two large packages like the previous chapter describes. In the code no user-oriented distinction between the user interfaces IconNet and IconMessenger is maintained. The code applies a functional difference between reusable JUNG extension classes and domain specific classes. The latter use iconic concept terminology or Peircean relations. An overview of the framework is given in subsection 5.2. The IconMessenger is a stripped version of the original application Lingua (by Fitrianie) extended with synchronization, so that messages simultaneously can be sent and received.

The way the source code is described is using implementation guidelines. Important pieces of code are highlighted when certain programming topics are explained. Topics like how to browse the iconology, how to change the way grammar is parsed, how to layout the graph, etcetera are described in subsection 5.3.

The framework originates from the application Lingua although not much of the original files still exist. Some information about porting the previous applications to the contemporary one can be found in appendix C.

The programming language Java is used. Version 1.4 of the Java Runtime Environment is sufficient to run the software. The software can be ob-

tained from http://sourceforge.net/projects/vilaug or http://vilaug.sourceforge.net. Java is an interpreted language, so the software is platform independent (except for the fact that Java has to be installed). The software may also be programmed in a way that enables running in a Java applet[1] in the browser. Familiarity with programming languages in general is assumed for this chapter.

5.2 Package Overview

The framework contains two large packages. The VilAug package (vilaug) and the JUNG Extension package (jung.ext). The latter contains all kind of files that are extensions upon the JUNG package. The VilAug package contains classes that do not derive directly from existing JUNG classes. These packages can however not be used independently from each other. There are mutual requirements. The JUNG Extensions are often required for classes in the VilAug package. The following package names exist:

- jung.ext: the JUNG Extension package:
  - jung.ext.actions: defines a set of action items that can be put in menus etcetera and are tied to defined mutations (see subsection 5.3.2 about how to implement a user action);
  - jung.ext.dag: adds DAG restrictions and defines methods to browse the graph (see subsection 5.3.1 about how to browse the iconology);
  - jung.ext.elements: adds a factory that creates the right type of graph elements (vertices, edges), a checker that checks for acyclicity for example and a dialog that asks for confirmation, additional data or offers information;
  - jung.ext.graphml: defines default GraphML elements, creates processors for the elements (see subsection 4.10.2 with the grammar rules API);
  - jung.ext.icon (see subsection 5.3.7 how to enrich icons with additional properties);
  - jung.ext.mutations (and jung.ext.mutations.mutators): defines mutations upon the graph, defines mutation modes (with sets of mutations), defines mutators (that execute the mutations) (see subsection 5.3.2 about how to use a mutator in a user action);
  - jung.ext.predicates (and jung.ext.predicates.general, jung.ext.predicates.edge, jung.ext.predicates.edge.impl, jung.ext.predicates.vertex, jung.ext.predicates.impl): establishes ways to filter or exclude vertices and edges upon type, parents (see section 4.9 about filtering the graph);
  - jung.ext.registry: adds registry to store all kind of references to factories, dialogs, etcetera, contains also a controller to complete the Model-View-Controller [MVC] design;
  - jung.ext.utils: convenient graph, string and set handling routines, basic interfaces;

[1] a demo is available at the aforementioned webpage

- jung.refact: the JUNG Refactoring package;
- thd: third-party objects;
- vilaug: the VilAug package:
  - vilaug.components: dialogs and panels for login, logout, icon input, conformations, the icon menu;
  - vilaug.exceptions: a collection of exceptions that can be raised;
  - vilaug.grammar: defines grammar constituents (like rule, rule element) and stores the grammar parser (contains the GrammarModule, see subsection 5.3.9);
  - vilaug.icon: defines icon properties, defines functions that use browsing functionality (contains the IconModule, see subsection 5.3.8);
  - vilaug.io: defines search paths, finds files and icons (contains the IOManager);
  - vilaug.peirce (and vilaug.peirce.mutators): mutating functionality that is in the realm of peircean relations (see section 4.11).

As this list shows, two tiny packages are not mentioned until now. The JUNG refactoring (jung.refact) package contains a collection of convenient refactored files. Most often refactored with certain class extensions in mind. For example the EditingGraphMousePlugin is adapted so that mousePressed(vertex) is called when the user clicks on a certain vertex. The third-party package (thd) contains some additional files of third-parties. In this case classes that perform a permutation of the elements in a grammatical rule (for the grammatical module).

There is too much code to be discussed in a decent manner. Therefore a particular approach will be taken. The code will be discussed from the viewpoint of a potential developer.

5.3 Implementation Guidelines

The following subsections guides a potential developer of the framework in implementing certain specific behaviour. It willl become clear how to add a certain user action, how to add a mutation type, etcetera. Some caveats are mentioned along the way.

5.3.1 How to Browse the Iconology

In the previous chapter the modelled classes and some methods are discussed. However, it might be still unclear how the ontology will be actually queried. The module that defines a lot of these functions is the already mentioned IconModule. A module is an interface in the VilAug framework. The default implementations are called managers. The corresponding IconManager contains the functionality needed and is aware of the graph properties of the ontology. There is for example the function getIconChildren, see algorithm 2.

Firstly, the icon representation is converted to a vertex representation. Secondly, there are limits set (as explained in subsection 4.8.3) that limit the search

Algorithm 2 Retrieve Icon Children from the IconManager

```
public Set getIconChildren(VilAugIcon parentIcon, VilAugIcon domainIcon) {
    DefaultIconVertex parent = (DefaultIconVertex) ((VertexIcon)parentIcon).getVertex();
    DefaultIconVertex ancestor = (DefaultIconVertex) ((VertexIcon)domainIcon).getVertex();
    parent.setElder(getVertex(ApplicationVerboCentro.ROOT));
    parent.setElder(getVertex(ApplicationVerboCentro.TOPOLOGY));
    parent.setElder(getVertex(ApplicationVerboCentro.ONTOLOGIES));
    parent.setElder(getVertex(ApplicationVerboCentro.GRAMMARS));
    Set icons = parent.getChildren(ancestor, parent);
    parent.removeLimits();
    return IconGraphUtils.vertexSetToIconSet(icons);
}
```

by adding vertices where the search should not continue. The limits have to be removed after each query with removeLimits. Thirdly, there is a reformulation of terminology. The term domain becomes ancestor. Fourthly, a function from the IconGraphUtils class is used to transform the vertex representation back again to the icon representation.

This is a typical example of the way the hierarchy is browsed or queried. Another implementation detail that involves the visualization icon. This icon stands for the way the icon menu is browsed. It can be read as "visualization manner in regard to browsing the icon menu". It is actually the same as the way icon children are returned. There are currently two visualization manners, one descending the icon menu along grammatical relations, one descending it along ordinary relations. So, in the former case the icon menu has the grammatical categories, like "actor", "patient", "instrument", in the latter case, it has the categories "living world", "sport and entertainment", "culture and religion", "science" for example.

5.3.2 How to Implement a User Action

The user initiates all events regarding data changes (except for loading the data from the files). These user intentions are conveyed by mouse movements and key presses. These movements, or *user actions*, are directly executed in a graphical environment or chosen in a menu. The items in such a menu are called *abstract actions*. If a class does have a name with "action" the latter is most often meant. This subsection starts with the editing plugins. It continues with the way in which a new user action and a corresponding mutation listener (mutator) can be defined. It ends with the receiveing instance, the VisualizationModel, that finally applies the mutation to the graph.

5.3.2.1 Editing Plugins

The classes that react upon the user or provide items in the form of a menu, are "graph mouse plugins". In the refactored package can the EditingGraph-MousePlugin and the EditingPopupGraphMousePlugin be found. The former reacts upon events like mousePressed(Vertex vertex) and mouseDragged(Vertex source, Vertex target). The latter creates a menu with the mentioned actions. The graph mouse plugins are collected in an EditingModalGraphMouse class. The graph mouse class can work in modes, what makes it possible to have a mode with only viewing and browsing functionality, another with editing functionality, etcetera.

The popup mouse plugin uses an ActionFactory. This class is an encapsulated MutationFactory, and provides AbstractAction objects instead of Graph-Mutation instances. The abstract actions can be added to a swing JPopupMenu. The users chooses subsequently an item. And the corresponding mutation is evoked as usual.

Adding a user action involves subclassing or decorating several classes. The plugins are written in such a way that extension is very easy. For example, the event methods that are called automatically have graph elements instead of just positions as their arguments (see as example algorithm 3).

Algorithm 3 GraphMouse Plugin & Vertex Arguments

```
protected void mouseDragged(Vertex source, Vertex target) {
  if (mode == EditingMode.REFERENCING) {
    referenced((IconVertex)source, (IconVertex)target);
  mode = EditingMode.INTERPRETING;
  }
}
```

This code stems from the EditingPeirceGraphMousePlugin and like can be seen, it uses a similar modular approach. It defines the function referenced(IconVertex representamen, IconVertex referent), that has IconVertex objects as arguments. And it adds semantics usuable in the extended class.

The user action depends on (besides mouse movements and clicks) mouse and key button combinations. Combinations of these buttons are stored in the MouseMode class. They are changed for a plugin as in algorithm 4.

Algorithm 4 GraphMouse Plugin & Buttons

```
EditingGraphMousePlugin editPlugin = new EditingPeirceGraphMousePlugin();
  MouseModes modes = editPlugin.getMouseModes();
  modes.setMode(MouseModeName.PRESS, MouseMode.CTRLBUTTON1);
  modes.setMode(MouseModeName.DRAG, MouseMode.CTRLBUTTON1);
  modes.setMode(MouseModeName.RELEASE, MouseMode.CTRLBUTTON1);
graphMouse.setEditingPlugin(editPlugin);
```

The code in algorithm 4 comes from the IconNetPanel. With this code are the functions that react to pressing, dragging and releasing mouse buttons now only executed when the control key is pressed simultaneously. It is also possible

to define other MouseModeName types so that the plugin reacts upon several types of key and mouse button combinations depending on the situation. In the IconNetPanel this is necessary because there is also double-click and popup functionality defined upon vertices.

5.3.2.2 New Action

An entire new user action involves however much more than only defining the way it is evoked by the user. The user action itself does have content that is encapsulated in a GraphMutation object in the framework. So, let us assume that we want to define a new abstract action, like “change icon”, that will be evoked at vertices. This means that the AbstractMutationAction class has to be overloaded like in algorithm 5.

Algorithm 5 ChangeIconAction

```
public class ChangeVertexAction extends AbstractMutationAction {
  private Vertex currentVertex;
  public ChangeVertexAction(Vertex vertex, ActionFactory actionFactory) {
    super(”Change vertex”, actionFactory);
    currentVertex = vertex;
  }
  protected void createMutation() {
          mutation = ((CertainMutationFactory)getActionFactory()-
.getMutationFactory()).changeIcon(currentVertex);
  }
}
```

It is of course often the case that a specific mutation factory has to be used. One that not only knows how to create, remove and alter vertices and edges. One that knows how to change an icon like in this example (see algorithm 5). This mutation factory has to be defined and implemented next. It contains in this case a function changeIcon(IconVertex vertex) that returns a GraphMutation. That factory can use a new or old element dialog and eventually an element checker. The mutation is now initialized and send the controller that notificates all its mutation listeners called mutators (see subsection 4.6.2).

5.3.2.3 New Mutator

The mutators all derive from AbstractMutator. The content of a mutator can be minimal, some just transmit the mutation to the model. But they can also be pretty sophisticated. One of the larger mutators can be found in the Peirce package, see algorithm 6.

The activate icon mutator in algorithm 6 uses a mutation factory itself. The activate icon mutation causes a cascade of mutations. The activate icon involves the creation of two new peircean relations, that have to be preceded by removing certain edges. The model is in this case only used to return a new element by “biproduction”. It uses its vertex arguments to return a new vertex in a particular way. Subsequently a mutation can be initialized and broadcasted, or the method calls directly a function in the model, like with model.changeVertex(vertexPair) to change a vertex in another.

Algorithm 6 ActivateIcon Mutator in Peirce Package

```
//a lot is skipped in this example!
protected void performMutation(PeirceRelation relation) {
  //create new elements using the model
  newheart1 = model.applyBiproduction(representamen, heart);
   relation1 = new PeirceRelation(representamen, heart, interpretant, new-
heart1);

  //precede actual mutation by necessary mutations
  m1 = getMutationFactory().removeEdge(representamen_heart);
  m2 = getMutationFactory().removeEdge(heart_referent);
  m3 = getMutationFactory().removeEdge(interpretant_heart);
  controller.mutate(m1); controller.mutate(m2); controller.mutate(m3);

  //performing actual mutation
  m4 = getPeirceMutationFactory().createRelation(relation1);
  controller.mutate(m4);
}
```

5.3.2.4 VisualizationModel

The VisualizationModel contains functions like addVertex(Vertex vertex) and removeVertex(Vertex vertex). See algorithm 7 for the code to add a vertex.

Algorithm 7 CreateVertex in VisualizationModel

```
public void addVertex(Vertex vertex) {
  stop();
  getMutationLayout().getOriginalGraph().addVertex(vertex);
  getVertexVisibilityPredicate().update();
  getMutationLayout().updateAddition(vertex);
  restart();
}
```

In algorithm 7 can be seen that in the method addVertex the layout algorithm is stopped. The main event is next: the vertex is added to the retrieved graph object. A predicate is updated to make the new vertex also visible and the layout is updated to get also a position for the new vertex (and adjust the positions of the older vertices). At the end the layout is started again.

The specific details of other user actions and mutation types are not handled in this subsection. The commentary in the source code documents handles this in much greater detail.

5.3.3 How to Display the Iconic Concepts

Like in subsection 4.4.2 is explained is the iconic representation of the icon/vertex object encapsulated in the VertexIcon interface. The VilAugIcon interface can be used in the case the reference to the corresponding vertex can be omitted altogether. The properties of such a VilAugIcon object are described in subsection 5.3.3.1 and the visualization of the graph and its elements on the screen is

described in subsection 5.3.3.2.

5.3.3.1 Iconic Concept Properties

The VilAugIcon interface and the interfaces it extends, have methods to set tooltip information, to reference a picture, to scale the icon and to cascade the icon. See algorithm 8.

Algorithm 8 VilAugIcon Properties

```
//commentary omitted and ownership simplified
public interface VilAugIcon {
  //tooltip setter and getter
  public void setTooltip(String tooltip);
  public String getTooltip();

  //current picture reference
  public String getFileName();

  //addition and deletion of cascaded icons
  public void add(Icon icon);
  public boolean remove(Icon icon);

  //scale icon
  public void setScaledSize(int width, int height);
}
```

It is possible to add additional functions to this interface or to subclass this interface. An addition method like for example turnOnAnimation, should be recognized by the proper classes. The IconButton might be reacting upon certain mouse behaviour and call this method on the VilAugIcon instance it contains. The data that belongs to additional properties stems by default from data in the vertex representation of the icon. Information about that subject can be found in subsection 5.3.7.

5.3.3.2 Visualization with Predicates and Decorations

A functional class that limits the amount of elements that are visualized is the VertexVisibilityPredicate that is used by the EccentricLayout. The way this layout works is explained in subsection , but now is it sufficient to know that it only shows a small subgraph and not the entire graph. This predicate is adapted when the user adds graph elements or on double-click at a vertex. The children of the vertex will in that case also pass the predicate and get a position from the layout.

Most of the visualization matter is done in the original JUNG package. The decorating class IconVertexIconAndShapeFunction knows the icon that belongs to a certain vertex and also returns a corresponding shape. This shape defines the area where the item can be clicked and froms the border of arriving or departing edges. This VertexIconAndShapeFunction can be added to the PluggableRenderer together with a lot of other graphical decorators. Like for example there is the EdgeShapeFunction. A TooltipListener is added to the

VisualizationViewer that contains other decorators. Like for example a post-renderer for graphical information at the background, Using other tooltips, or adapting the shape of vertices is done by adaption of the plugin. An example to adapt the shape of vertices is shown in algorithm 9.

Algorithm 9 IconVertexIconAndShapeFunction

```
//functionality omitted from the example!
public Shape getShape(Vertex v) {
  if (vertex.getType() != IconVertexType.INTERICON) {
    return super.getShape(v);
  }
  VertexShapeFactory vsf = new VertexShapeFactory();
  return vsf.getEllipse(v);
}
```

The mentioned JUNG classes are like plugins and can be plugged in and out the PluggableRenderer at wish.

5.3.4 How to Layout the Graph

Layouts come in differen flavours. There are layouts that try to minimize certain parameters by iteration of vertex positions. This algorithm can adjust positions using criteria like edge crossings, or vertex relateness, or vertex parentship. There are also layouts that calculate the vertex positions using some parameters and do not need to iterate vertex locations, but display the vertices directly at the right positions. The latter layout is in JUNG terminology a StaticLayout and is used for our purposes because it gives a quiet screen where vertices do not drift away unintentionaly.

An abstract layout class, the MutationLayout, contains several predefined methods.

Algorithm 10 shows a lot of methods in the layout aimed at vertices. There are corresonding functions for edges. The vertices have however positions assigned to them. The edges are drawn between vertices with a certain shape, and contain no location information. Visibility is defined by a predicate set in the setVertexVisibilityPredicate method. The original graph can be accessed by getOriginalGraph, the visualized vertices by getVisualizedVertices. The layout works with a copy of the original vertices, these can be obtained by getVisibleVertices. This enables temporarily adding position fields and other layout information to vertices, without them being saved to the file when the original elements are stored for example.

The initialization methods are separated in several subfunctions. The initializeVertices functions takes all visualized vertices and calls initializeVertexLocation for each of them. That function suggests a position for a vertex. Subsequently the initializeVertex(Vertex v) is called that enables a subclass like EccentricLayout to update the position or other data according to its metrics.

There are also several update methods. The most general method is just called update. All visualized elements are recalculated. The method updateVertices recalculates only all visualized vertices. When a certain kind of mutation is known, it is not necessary to recalculate all vertex positions. When a vertex is

Algorithm 10 MutationLayout Methods

```
//general functions in MutationLayout class
public abstract class MutationLayout {
  //filtering elements in regard to visibility
  public void setVertexVisibilityPredicate(VertexPredicate p);
  public Graph getOriginalGraph();
  public Set getVisualizedVertices();
  public Set getVisibleVertices();

  //layout intialization
  protected void initializeVertices();
  protected void initializeVertex(Vertex v);
  protected void initializeVertexLocation(Vertex vertex);

  //update layout
  public void update();
  protected void updateVertices();
  public void updateAdditions();
  public void updateAddition(Vertex vertex);
}
```

added to the graph, only the neighbourhood of the new vertex might be recalculated, for example. Hence, the updateAddition methods. For information about which additional elements the layout has to visualize exactly, the vertexPredicate.getOutputChange can be used. See subsection 4.9 about the properties of the VertexPredicate object.

The actual calculation is done in the EccentricData object. This is a field with parameters that decide the position of a vertex. It defines boys and girls. The former are the relational concepts that will appear in an inner circle around their parent. The latter are normal concepts that will appear in an outer circle around their parent. This is already described in section 4.7. The way this algorithm can be altered is by adjusting variables like the radius of the circles involved, or the way the radius is calculated. See algorithm 11.

Algorithm 11 Calculation of Radius in Layout

```
public double getRadius() {
  if (gender == Gender.FEMALE) {
      return ((getLevel() + 1) * Math.cos(getAngleIncrement() / 3)) * RA-
DIUS_RATIO / (double)1;
  } else if (gender == Gender.MALE) {
      return ((getLevel() + 1) * Math.cos(getAngleIncrement() / 3)) * RA-
DIUS_RATIO / (double)2;
  }
  return 0;
}
```

Like can be seen in algorithm 11 is the level parameter used together with an angle parameter to decide how big the circles becomes.

5.3.5 How to Show a new Dialog

Dialogs and panels are provided by a general instance, the PanelFactory, like briefly mentioned in subsection 4.3.1. The PanelFactory contains a reference to the graph, so that the buttons and menu items in dialogs (like OK, CANCEL and FILE LOCATION) can be retrieved from the iconology itself. The PanelFactory is able to provide several panels and contains a mode. See algorithm 12.

Algorithm 12 PanelFactory Contract

```
public interface PanelFactory {
  //panel types that can be returned
  public DialogPanel getDialogPanel();
  public JPanel getHintPanel();

  //factory mode
  public void setMode(DialogMode mode);
  public DialogMode getMode();
}
```

The DialogMode methods indicate to the PanelFactory what kind of panel can be returned. The DialogPanel is a JPanel extension, that adds a title and a routine to check if the data entered by the user is correct. The DialogPanel should be appended by OK and CANCEL buttons. The getHintPanel method returns a normal JPanel with only text. No buttons are necessary. The PeircePanelFactory uses the following DialogModes: INPUT, CREATION and CONFIMATION. Each of these modes result in another DialogPanel.

So, an entirely new dialog can be shown by adding a DialogMode and corresponding DialogPanel in the PanelFactory. Every unit that has access to the PanelFactory, such as an ElementDialog can subsequently set this mode and retrieve the appropriate DialogPanel.

5.3.6 How to Run as an Applet or Application

The application can run as an applet and as an application. To achieve has the VilAugMain class the two functions main and init. The former is evoked when it runs as an application, the latter when it runs as an applet. To code is archived in a .jar file and is made self executable with a manifest file. The content of this file tells the java interpreter where to find the other files, see algorithm 13.

Algorithm 13 Manifest File

```
Manifest-Version: 1.0
Main-Class: vilaug.VilAugMain
Class-Path: data.jar icon.jar commons-collections-3.1.jar jung-1.7.2.jar kxml.jar
```

The actual creation of the .jar file is very easy with the aid of a (DOS) batch file, see algorithm 14.

The source files are stored in several subdirectories of the \bin directory.

The information in algorithm 13 resembles what is written to reference the same libraries in the case of running as an applet (see algorithm).

Algorithm 14 Creation of Self-Executable Jar

```
jar -cfm ..\run\vilaug.jar MANIFEST.MF -C ..\bin\ jung\ -C ..\bin\ thd\ -C
..\bin\ vilaug\
cd ..\run
java -jar vilaug.jar
```

Algorithm 15 Applet Evokation

```
<APPLET type="application/x-java-applet;version=1.4"
  width="640" height="440" align="baseline" code="vilaug.VilAugMain"
  pluginspage="http://java.sun.com/j2se/1.4/download.html"
    archive="jung-1.7.2.jar, vilaug.jar, data.jar, commons-collections-3.1.jar,
kxml-min.zip, icon.jar">
 No Java 2 SDK, Standard Edition v 1.4 support for APPLET!!
</APPLET>
```

The actual code uses actually a slightly other approach to account for different browsers, but this is the way the applet should be initiated. In the two described ways becomes the framework up and running.

5.3.7 How to Enrich the Iconic Concepts with Additional Data

It may be desirable to add a type of data to iconic concepts that is not defined currently. Like translations, a URL to an online resource or frequency information. Several things have to be done to handle such a new property of an iconic concept. First of all a new key has to be defined in the top of the GraphML file. See algorithm 1 for the GraphML API.

The GraphML units should recognize this new type and have corresponding java classes running. This information is probably used in the visualization of an icon. Like for example a parameter that adds an explanatory gesture in a certain sign language as additional information to a VilAugIcon object. See also subsection 5.3.3 about the properties of the VilAugIcon interface. The first class that has to be adapted in the GraphML package, is an implementation of the GraphMLElement interface. In the GraphMLFileReader are the data and keys stored and upon creation of a graph element, is this information passed to the GraphMLElement object. In the IconGraphMLElement class is a good example of the way these information is read.

Adding additional data involves the creation of an additional key. The new type has to be added in the procedure of algorithm 16. A new IconValue instance has to be defined in that case, like IconValue.ANIMATION. Until now the information is parsed correctly. Subsequently, the ElementFactory (subsection 4.3.1) will use it to actually create an icon element (and with that a vertex element). See algorithm 17.

This icon is subsequently used in the createVertex method and will be available in the framework. A graphical or other component may use it when appropriate.

Algorithm 16 Reading Graph Element Data

```
//check key restrictions, and transform to convenient type
protected Map getContent(UserDataContainer element, Set data, Set keys) {
  //initialization, load current data and key objects from set

  //search proper key for current data object
  if (key.getID().equals(datum.getKey())) {
    //check if key is defined for the proper element (edge, vertex)
    if (key.fitsDomain(element)) {
      //put all known properties in a contents map
      if (key.getName().equals(IconValue.FILENAME)) {
        contents.put(IconValue.FILENAME, new IconContent(datum.getContent()));
      } else if (key.getName().equals(IconValue.TOOLTIP)) {
        contents.put(IconValue.TOOLTIP, new IconContent(datum.getContent()));
      } else if (key.getName().equals(IconValue.VISIBILITY)) {
        contents.put(IconValue.VISIBILITY, new IconContent(datum.getContent()));
      }
    }
  }
}
```

Algorithm 17 Applying Graph Element Data

```
//use properties to create or decorate an icon element
public void setCargo(Object cargo) {
  //load information
  Map contents = (Map)cargo;
  IconContent filename = (IconContent) contents.get(IconValue.FILENAME);
  IconContent tooltip = (IconContent) contents.get(IconValue.TOOLTIP);
  IconContent visibility = (IconContent) contents.get(IconValue.VISIBILITY);

  //create icon element
  VertexIcon icon = new DefaultIcon(filename.toString(), tooltip.toString());
}
```

VilAug IconMessenger IconNet

5.3.8 How to Store the Iconology in a Different Way

The iconology is accessed using the general interface IconModule. It contains functions like getIconChildren as discussed in subsection 5.3.1. The IconModule formulates the contract that has to be obeyed by its implementation.

Algorithm 18 IconModule Contract

```
public interface IconModule {
  //general iconic concepts like ontologies and grammars
  public VilAugIcon getTopology();
  public VilAugIcon[] getOntologies();
  public VilAugIcon[] getGrammars();
  public VilAugIcon[] getVisualizations();

  //checking routines
  public boolean isGrammar(VilAugIcon icon);
  public boolean isOntology(VilAugIcon icon);

  //browsing and querying routines
  public Set getIconChildren(VilAugIcon parentIcon, VilAugIcon domainIcon);
  public boolean isIconTerminal(VilAugIcon icon, VilAugIcon domainIcon);
  public VilAugIcon[] getMenuItems(VilAugIcon domainIcon);
 public VilAugIcon getGrammarType(VilAugIcon icon, Set potentialGrammar-
Types);
 public VilAugIcon getVisualizedIcon(VilAugIcon domainIcon, Set potentialVi-
sualizations);

  //file and graph dependent routines
  public IconGraph getGraph();
  public void setEditableGraph(String fileName);
  public void saveGraph();
}
```

The IconModule should return general iconic concepts, like the top of the topology, a set of icons that are the ontologies, a set of grammars, and a set of visualizations. In the context of an icon menu is a visualization manner important. If the iconic concept CAT is used, it will probably be interpreted as browse the icon menu in a cat-like manner. By default does the IconManager only return the iconic concepts GRAMMAR and ONTOLOGY as visualizations. It is possible to expand this with the iconic concept ORTHOGRAPHY, or LOCATION. With this a certain specific orthography, respectively, certain localized icon representations are preferred in the visualization.

Most interesting are the browsing and querying routines. The getIconChildren function returns the children of an item in the icon menu. The isIconTerminal function indicates the fact that an icon has no children anymore. The domain is given by an icon, in practice a certain ontology. The menu items are options that appear on the menu. The options include switching from grammar, ontology or visualization for example. The function getMenuItems given the GRAMMAR domain, returns all icons that are indicated as menu-items. In the iconology this is implemented by having the icon MENU-ITEM as ancestor of

several grammars. Grammars that should not appear in the option menu, are not connected to the MENU-ITEM. An implementation should return all menu items that are indeed usable.

The getGrammarType and getVisualizedIcon return the appropriate item from a given set of available options. The former returns for example the icon INSTRUMENT CASE when HAMMER is given as an argument, and a set of grammatical cases from case grammar. The latter returns for example the icon CASE GRAMMAR when the domain GRAMMAR and a set with CASE GRAMMAR, ENGLISH GRAMMAR, TOURIST ONTOLOGY and SUMO UPPER-LEVEL ONTOLOGY is given. Like can be seen are all queries performed with iconic concepts as arguments. This approach is recommended in other implementations of the IconModule too.

The file and graph dependent routines are not entirely discoupled from the graph representation. The distinction between the data representation and the icon module is therefore not complete. Another implementation of the IconModule should obtain file modularity in some similar way. For further information and acutal code is the IconManager and corresponding javadoc commentary the proper place to look.

5.3.9 How to Parse Grammar in a Different Way

The general interface called the GrammarModule discussed in subsection 4.10 does have the default implementation called the GrammarManager. The manager knows the mechanics to weave a grammar graph from a set of rules. If the way the grammar is parsed should be changed, another implementation of the GrammarModule can be used. The contract as defined by this module has to be obeyed.

Algorithm 19 GrammarModule Contract

```
public interface GrammarModule {
  //decoration
  public void setIconConverter(IconConverter converter);

  //main function
  public Set nextTerminals(VilAugIcon grammarIcon);

  //initialization
  public void setGrammars(VilAugIcon[] grammars);
  public void loadGrammar(VilAugIcon grammar);

  //visualization
  public void visualizeParsing();
}
```

The GrammarModule defines an IconConverter. This class knows how to convert GrammarVerboCentro objects to VilAugIcon objects. The former are strings explicitely defined in a grammatical context. This converter enables the GrammarModule implementation to understand the VilAugIcon objects and return them. The nextTerminals function is the most important function. It receives an icon and returns the first item of possible - grammatical correct -

sequences. A new rule is indicated by calling it with null. The initialization functions are to enable loading the data from files. The setGrammars method passes the icons that are grammars. And in a specific order such that dependencies between files are obeyed. The loadGrammar method takes care of one these grammars. This grammar will be used when nextTerminals is called. The visualization method shows by default a graph.

The grammar can be parsed differently by defining an altogether different GrammarModule implementation. It is also possible to adjust only certain functions in the grammar module, or adjust only some facets of the parsing algorithm. For example the use of two different parsers, one for unordered, one for ordered languages. Or to adjust the parser so that it knows to handle context-sensitive grammars. Or to optimize speed. Or to provide another visualization (in another form than a graph for example).

To be able to handle unordered grammars currently two third party classes are used, Perms and PermGen. These classes from an extension op de java Iterator. Each time next() is called, a new permutation of a specified set will be returned. All these permutations are subsequently fed into the parser. This is the reason that rules of only three or less elements are recommended in an unordered grammar. This is an example where an adapted implementation of the GrammarModule can turn out to be convenient.

5.3.10 How to Add an Entire New Language

A new language in the framework contains icons - normal, relational, grammatical icons - and grammar rules. The former are stored in GraphML files. The latter in a separate XML file. The application does not automatically include new data files. To add a language a child has to be added to the ONTOLOGIES icon and if appropriate to the CUSTOM GRAMMARS icon.

The information of a new language should be stored in separate files. References to these files are written down in the “vilaug.xml” file (part of the “data.jar” container). The syntax is as described in algorithm 20.

Algorithm 20 File Reference Syntax

```
<application>
  <files type=”ontology”>
    <file><name>certain_ontology.xml</name></file>
  </files>
  <files type=”grammar”>
    <file><name>some_grammar.xml</name></file>
    <file>
      <icon_id>certain grammar</icon_id>
      <name>certain_grammar.xml</name>
      <rules>certain_grammarrules.xml</rules>
    </file>
  </files>
</application>
```

The actual data access is gouverned by the IOManager. The class IOManager is used by all classes that have to access something from disk or the web. All

the functions in it are static and can be accessed with functions like getGrammarRules and getOntology. The "vilaug.xml" file is parsed by the IOInfoParser and in that way does the IOManager know which files can be loaded. The ontology and grammar files are loaded using index numbers. So the application does not have to know their names. The order in the "vilaug.xml" file is preserved. That makes it possible to store general concepts in default library files. The manager knows how to retrieve icons from a local disk or a server and is not dependent from the environment.

Chapter 6

Graphical Tour

The graphical tour describes GUI's, dialogs and user actions. At the same time stipulates it merciless the shortcomings of the current design. Positive as well as negative characteristics in regard to design practices as consistency, direct manipulation, feedback and closure are evaluated. An evaluation of the design can benefit from an indication about the time that will be needed to implement these issues. The usability analysis will be performed from the viewpoint of the developer [**?**].

This graphical tour starts with the IconMessenger in subsection 6.1, the IconNet in subsection 6.2. The user actions are described in detail in subsection 6.3 and the chapter ends with subsection 7.4 about recommended design focus for implementation sequences.

The VilAug framework does contain an IconMessenger and an IconNet. An introductionary screenshot of the tool IconNet can be seen in figure 6.1.

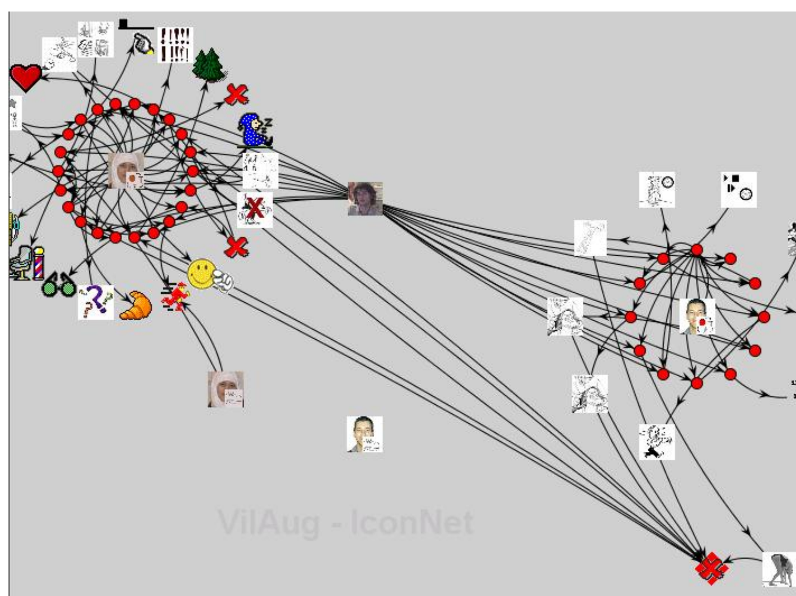


Figure 6.1: VilAug Introductionary Screenshot

The visualization throughout this chapter may vary. It is for example possible to adapt the size of the icons or to run the application with a metallic look and feel (default in later Java versions).

6.1 IconMessenger GUI

The IconMessenger does have the following layout in general, see figure 6.2.

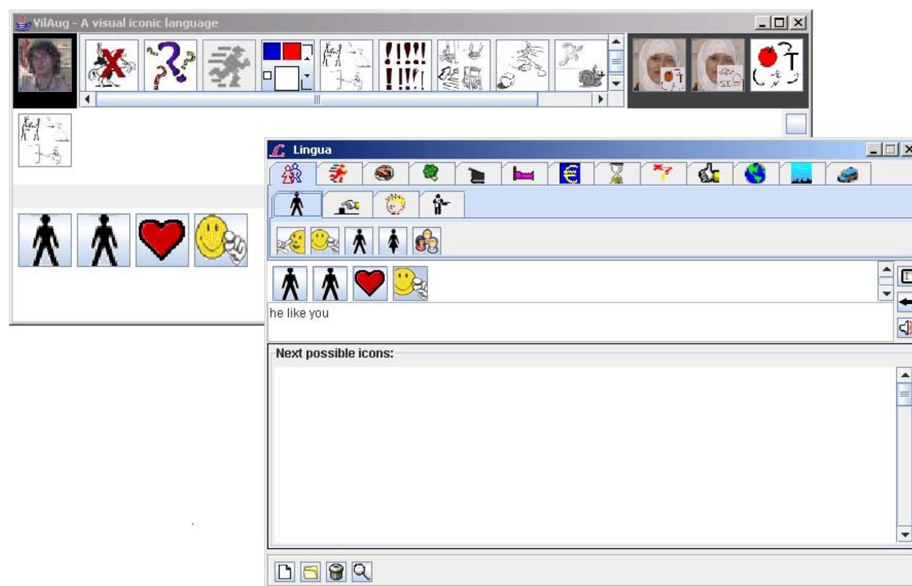


Figure 6.2: Typical icon messenger GUI (Lingua at the front)

The IconMessenger does have an icon menu panel at the top. At the right are menu options situated. From left to right is it possible to use these icons to select a grammar, an ontology and a visualization type. The panel in the center depicts all the icons on a certain level in the icon hierarchy (defined by the selected grammar and ontology. When the grammar disallows certain icons, they will be grayed out. The panel at uttermost left is the parent icon. By default an icon for the top of the topology is visualized in this corner.

The room beneath the icon menu, contains the icon message. It is formed by repeatedly selecting icons from the icon menu. The buttons at the right are to remove the last icon, the whole sentence or to send the message to another user.

The space beneath this is destined for additional information, frequency information etcetera. This is not implemented in VilAug, but functions from the previous application Lingua - like a translation module to a verbal expression, and predictions using bigram and trigrams - are good examples.

6.2 IconNet GUI

The vertices that are visualized in IconNet when it is shown for the first time are fixed. The available ontologies and grammars are among them. The graph can be scrolled by the scrollbars at the sides. Also can the satellite viewer at topleft be used to change the working area. When the mouse cursor hovers above a vertex, a tooltip appears. Browsing the IconNet is by (left) double-clicking vertices. See figure 6.3.

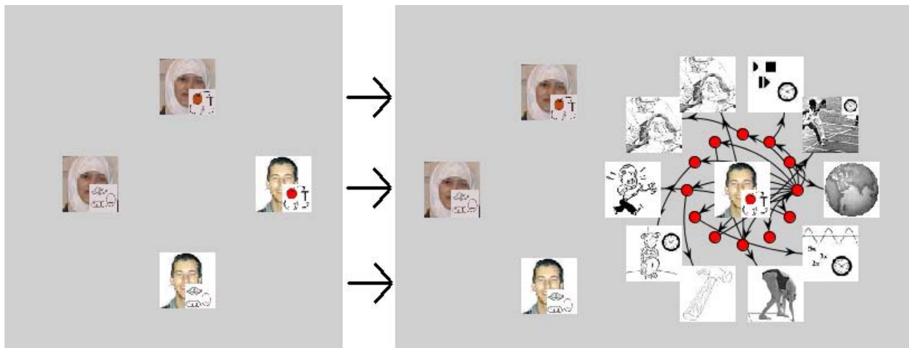


Figure 6.3: IconNet Browsing

Browsing is only one of the many actions the user has to perform. The mousebuttons have to be overloaded by extended modifiers (modifying keybuttons). With the control key combined with the left mouse button a popup-menu shows up. The items in this popup-menu depend on the situation. Above a vertex there is for example an option to remove that vertex, above empty space, there is an option to add a vertex.

There are also functions in IconNet that can be accessed without any popup-menu. The dialog for adding a vertex appears directly on left clicking in empty space. Relations can be drawn using the right mouse button.

6.3 User Actions

Two typical procedures will be handled in this subsection. Each of them exists out of several user actions. The first procedure is about adding a concept to a visual language. This involves the creation of an icon, and the creation of two relations. The second procedure upgrades a relation (like described in subsection 3.2.3).

6.3.1 Adding Concept to Visual Language

This subsection does does use one item as example. This concept has to be added to a visual language. The example does have the following content:

Example 35. A new icon, EUROPEAN HOUSE, is added to the visual language VIL, created by Leemans. It should appear in the top of the ontology (as unique beginner) and can be considered grammatically as belonging to the grammatical case “theme”.

VilAug
IconMessenger IconNet

The ontology and grammar of VIL are depcited by the icons LEEMANS' ONTOLOGY and LEEMANS' GRAMMAR (see subsection 3.2.1.1). The creation of the new vertex EUROPEAN HOUSE is with the icon input dialog. A screenshot can be seen in figure 6.4.

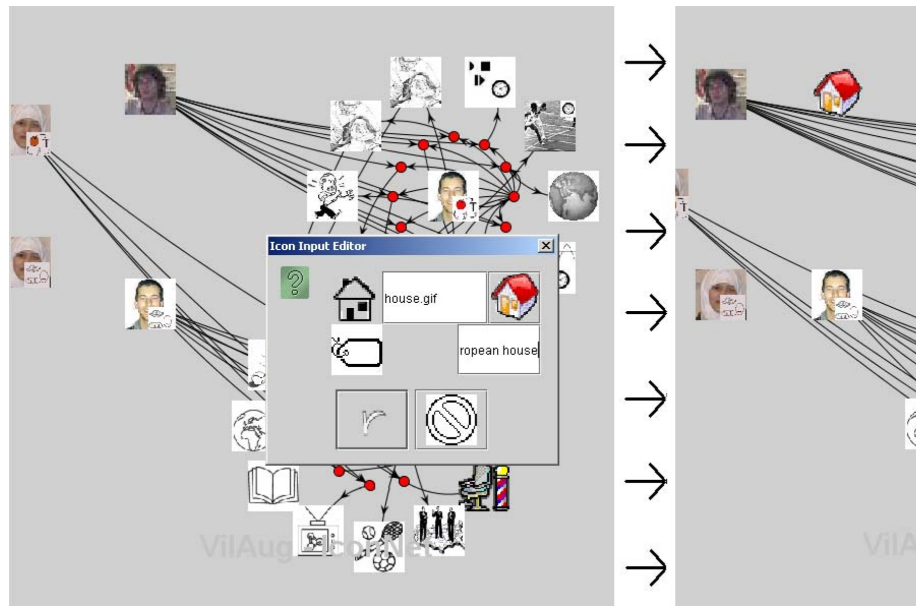


Figure 6.4: IconNet Icon Creation

The icon input dialog shows an edit box for the filename. This should contain the filename itself and its extension. The icon corresponding to this filename is visualized at the right top corner. It updates when the user clicks the icon. The user is allowed to enter a description in the second editbox.

The icon input dialog shows some anomalies. The layout is suboptimal, the icons are not updated automatically, the boxes do not have a convenient size and the icons should have a gray background. However, it also reflects some good design decisions. The icons for the okay and cancel buttons are taken - very consistently - from the iconology itself.

After the user confirms the entered data, is the vertex appended to the layout (and the graph). The new vertex is shown in the innermost circle of most important vertices. A separate location allocated to new vertices is another possibility. On exiting IconNet this area can be screened for vertices and the user asked about disregarding them or yet relating them to an ontology or grammar.

The relating procedure is already described in subsection 3.2.1.1. In practise does this mean the following: The user selects first the representamen, drags the mouse cursor to the referent, releases the mouse, and selects subsequently the interpretant. In figure 6.5 is this action captured in progress. The icon EUROPEAN HOUSE is attached to the top of the TOPOLOGY and LEEMANS' ONTOLOGY is its interpretant. Or in other words: EUROPEAN HOUSE is related with the LEEMANS' ONTOLOGY in the context of (the top of) the TOPOLOGY. This is like described in fact 32 of interchangeability. The order in which the interpretant versus the referent is clicked is not important.

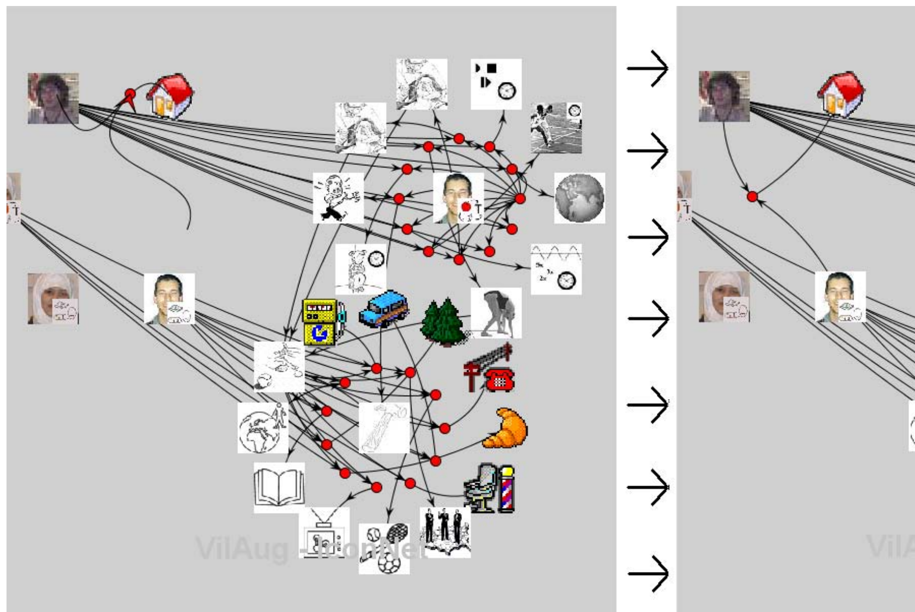


Figure 6.5: IconNet Relation Creation to Ontology

The results of the creation of a relation should in some way be visible in the icon menu. The icon european house is now indeed visible. It is however still disabled. See figure 6.6.

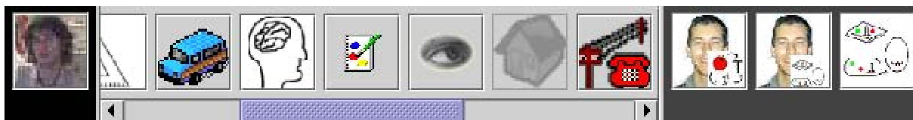


Figure 6.6: Icon Menu after Relation to Ontology

This is because no grammatical category has been chosen for it yet. To embed the icon entirely in the language, it has also to be connected to LEEMANS' GRAMMAR. This is done in a similar way as the creation of a relation to LEEMANS' ONTOLOGY. The grammatical cases "agent", "patient" or "theme", "source", "goal" and "instrument" have all merely all nouns as subclasses in VIL. In this report there will not be deviated from this approach. However, it is important to realize that having EUROPEAN HOUSE as an INSTRUMENT CASE doesn't make sense. There is no such sentence possible as "I paint the car with the house". Another point is that the icon EUROPEAN HOUSE can not directly be connected to the icon NOUN. When the icon functions as THEME CASE or as LOCATION CASE, the THEME CASE should not have NOUN itself as child, but the children of NOUN. Thus the icon NOUN should be invisible like can be done with the technique that is explained in subsection 3.2.1.4. The icon NOUN is an interface, and the icon EUROPEAN HOUSE is connected to ABSTRACT NOUN.

Figure 6.7 shows the EUROPEAN HOUSE related with the ABSTRACT NOUN in a LEEMANS' ONTOLOGY context. The relation uses again LEEMANS' ONTOL-

VilAug IconMessenger IconNet

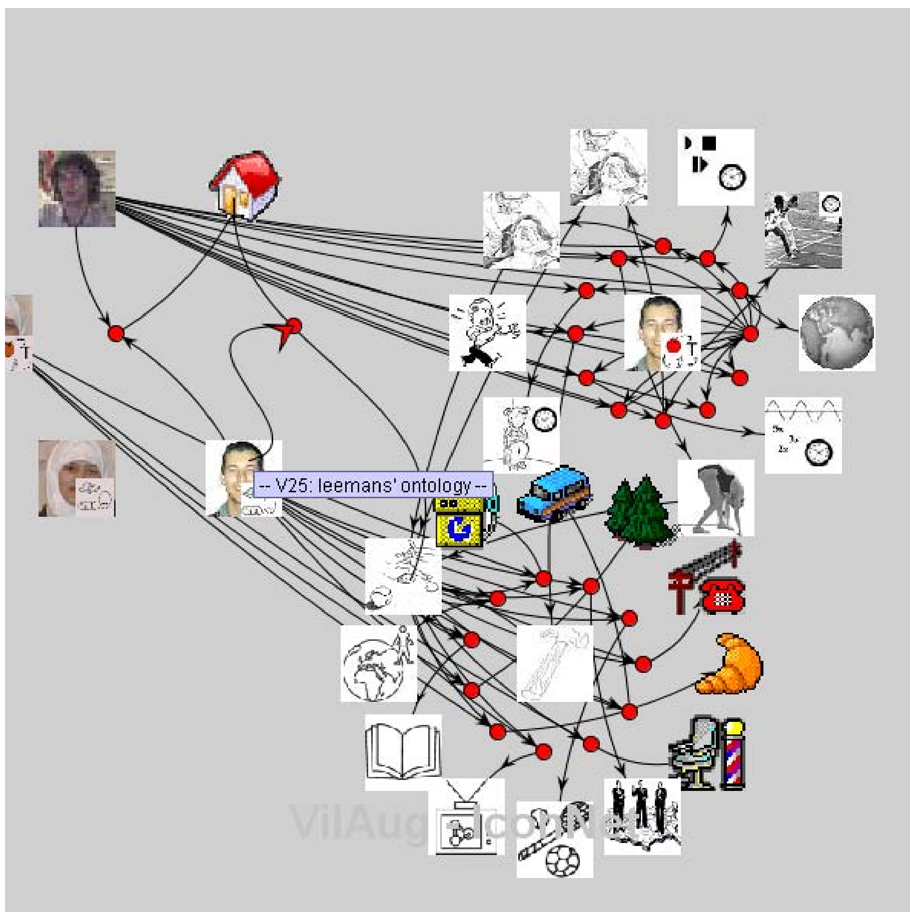


Figure 6.7: IconNet Relation Creation to Grammar

OGY. Currently not as representamen, but as interpretant. It seems perhaps redundant to have two references two LEEMANS' ONTOLOGY, but it is necessary. It is not predetermined that the EUROPEAN HOUSE is a NOUN. It is for example possible to have an ontology that indeed distinguishes between instruments, themes, etcetera, and not merely connects items to the general grammatical noun concept.

The entire example of creating a new icon, EUROPEAN HOUSE, and embedding it ontological and grammatically in the language VIL, is completed. The icon menu is now as follows:

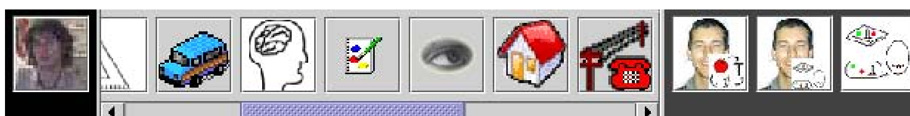


Figure 6.8: Icon Menu after Relation to Grammar

VilAug IconMessenger IconNet

Some remarks. There is an opportunity to bundle these actions together. This helps the user not to forget to assign a grammatical category to a new icon. Another point is that it helps the user to have a replica of the icon menu available in the IconNet environment. In that way the results of the user actions can directly be observed by the user.

6.3.2 Upgrading a Relation

The graph mutation "relation upgrade" is described in subsection 3.2.3. It can be seen as activation of an icon too. It involves selection of the representamen, referent and interpretant of a Peircean relation in the iconology. The relational icon is subsequently promoted to a normal (visible) icon. From the viewpoint of the relation, someone can also speak about specification. Again the same example is used. The upgrade action contains three subactions, bundled together by the framework in one big mutation event. A kind of macro. The content of this example is adjusted as follows:

Example 36. Upgrade the relational concept between the EUROPEAN HOUSE, the TOPOLOGY top, and the LEEMANS' ONTOLOGY, to an icon WESTERN SOCIETY, that instead of the EUROPEAN HOUSE is on the TOPOLOGY top and does have the EUROPEAN HOUSE as child in a Leemansian world.

To upgrade a certain relational concept, the user starts repeating a part of the icon addition procedure (see previous subsection 6.3.1). First a representamen, referent and interpretant are selected. This is followed by the same icon input dialog. With this dialog can data information about the to be upgraded relation be entered. See figure 6.9.

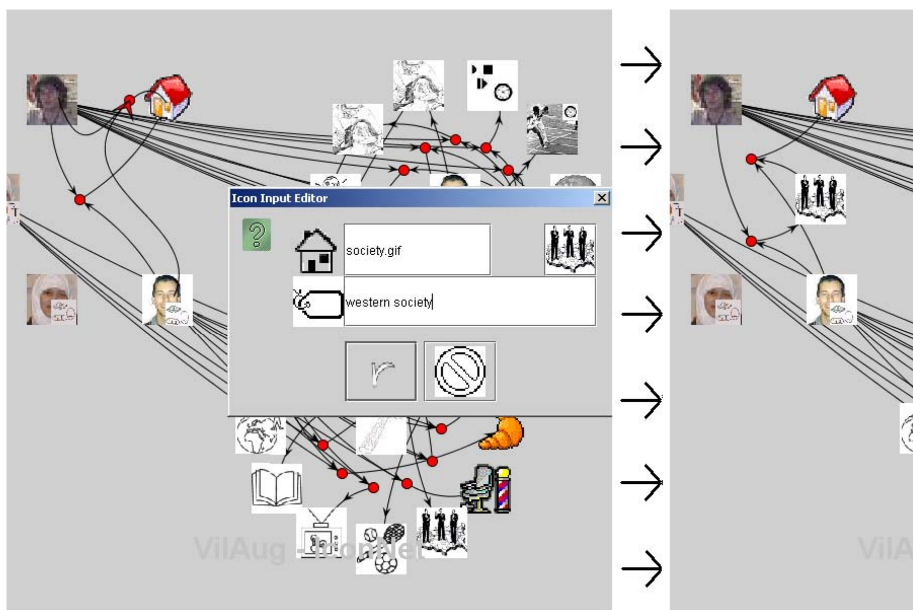


Figure 6.9: Relational to Normal Concept Creation

There is an important difference with the icon addition procedure. In this

VilAug IconMessenger IconNet

case is the order of selection of referent and interpretant very important. A reversed order will mean something different. When the order is reversed, the TOPOLOGY (top) is taken as interpretant. This means that both, the EUROPEAN HOUSE and the WESTERN SOCIETY will appear in the top of the topology. There is however, in that case no interpretation by LEEMANS' ONTOLOGY between WESTERN SOCIETY and EUROPEAN HOUSE. The icon menu will not show the EUROPEAN HOUSE icon at all in this case.

Upgrading a relational concept with LEEMANS' ONTOLOGY as interpretant works correct. The difference between the correct way of upgrading and the incorrect way is shown in figure 6.10.

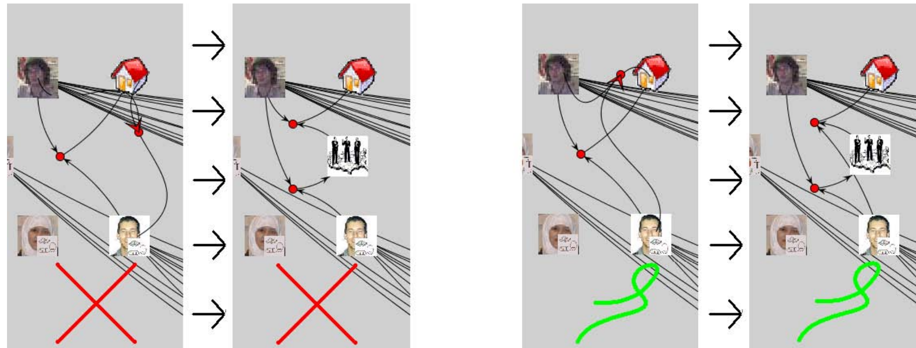


Figure 6.10: Upgrading Relation

The fact of interchangeability is still correct. After the user action has been performed every interpretant and referent can change roles. However, the "upgrade relation" itself uses a specific order of referent (second vertex) and interpretant (third vertex) to create edges in the graph. It is the interpretant that will be pointed to the new relational icons. The referent will still be related (by a relational icon) to one icon only, the new inserted icon. Swapping the roles of referent and interpretant leads to another ontology and another resulting icon menu. The final - correct - result in the form of the icon menu is as in figure 6.11.

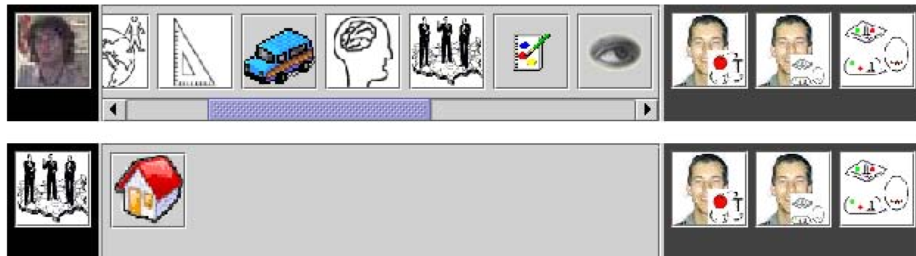


Figure 6.11: Icon Menu after Relation Upgrade

Figure 6.11 shows the icon menu two times. The menu at the bottom is obtained by clicking the WESTERN SOCIETY icon in the menu at the top.

A point of improvement is guidance in the upgrade relation mutation. Guidance in choosing the correct order of referent and interpretant will reduce the

amount of possible errors. The method of bundling actions in one mutation like in the upgrade relation procedure is convenient. From the point of interface design does it provide *closure* (see Shneiderman and Plaissant in[42]). Closure is organizing actions in groups that belong together. Adding beginning, middle, and feedback at the end, gives users the satisfaction of accomplishment. What can be improved in regard to closure, and other design principles, will be handled in the following subsection.

VilAug IconMessenger IconNet

Part IV

Evaluation

Summary Part IV - Evaluation The evaluation part contains one chapter (chapter 7) and a conclusion (chapter 8). The evaluation chapter starts with a graphical tour of the developed framework. It describes the strengths and weaknesses of the framework compared to other projects, like WordNet and CYC. It describes in what way the framework is an improvement upon the previous applications. It describes in what way the theoretical model appear to have negative consequences in regard to practical implementation. It provides a list of recommendations. A chapter with conclusions follows.

VilAug
IconMessenger IconNet

VilAug
IconMessenger IconNet

Chapter 7

VilAug Evaluation

This chapter contains evaluation material in several forms. Section 7.1 compares the project with WordNet and Cyc. Section 7.3 describes the added value in regard to previous work build upon in this project. Section 7.4 stipulates interface design points (encountered on the graphical tour from chapter 6). Subsection 7.5 describes impracticalities from the model in the implementation phase. Subsection 7.6 offers several recommendations.

7.1 Model Comparison

In subsection 2.2.3.3 about ontology types, are several characteristics mentioned about ontologies (see again [20]). The theoretical model that inspired a lot of design decisions in the software model, contains as main item, the ontology aimed at icons, the iconology. This iconology does have the following properties:

- Generally: the iconology is general-purpose and not restricted to specific domains. It is especially aimed at representing iconic concepts. It can integrate several domain ontologies. The amount of concepts is in the order of 10^2. Its formalism resembles conceptual graphs. It is implemented in the Java programming language and contains two visual languages as proof-of-concept. The iconology is published and freely available;

- Design process: the iconology does not predefine a way to build the iconic or relational ontology of a visual language. The language might be built bottom-up or top-down. By default it will the case that the concepts that are most easy to visualize by visual linguists or the ones that are most familiar to ontology engineers will become part of the iconbase. A specific visual language might be evaluated by means of usability metrics in regard to choosing icons from the corresponding icon menu. The implemented visual languages are evaluated in regard to their capability to convey unambiguously the meaning intended by the creator (by the inventors themselves). No evaluation in context of the iconology in choice speed rates, or other terms is performed;

- Taxonomy: several taxonomies can be stored simultaneously. The taxonomies can be from all kind and even mixed types. Each relation in a

taxonomy defines its type (hypernymy, holonymy, or other - also custom defined - types). In the iconology the basic building blocks like iconic concepts, relations and grammatical relationships are considered equally important. There is no treatment of time. There is no predefined top-level division, except for the concepts: TOPOLOGY, ONTOLOGIES and GRAMMARS. There is no quantitative value for the density of the iconograph, it is however the case that the amount of items should not exceed the short-term memory capacity of about seven chunks. The iconograph is not dense. It is however tangled, there is a lot of reusability possible and above that is multiple inheritance used;

- Internal structure of concepts: there is no internal structure of concepts. No properties are defined upon concepts. The roles of concepts are defined by a connection to a grammatical type in combination with grammatical rules over these types;

- Axioms: there are no explicit axioms. The iconology is a lightweight ontology, that contains a knowledge representation, no knowledge reasoning. There are therefore no axioms that function as the basis of a reasoning procedure;

- Inference mechanism: there is a kind of inference mechanism. There is for example a function that searches ancestry. This involves infering $A \rightarrow C$ from $A \rightarrow B$ and $B \rightarrow C$ if both use the same type (hypernymy, holonymy, etcetera). Reasoning that transfigures relation types to each other, like $A \looparrowright C$ from $A \rightsquigarrow B$ and $B \rightsquigarrow C$ is not embedded. The iconology goes beyond first-order logic, because relations are treated as first-class citizens. But again the iconology does not contain axioms, so the first-order logic issue is irrelevant;

- Applications: the iconology is an icon ontology used by VilAug, a framework that contains several visual iconic languages. Two user interfaces are provided. The IconMessenger uses the information from the iconology to retrieve the icons in a icon menu hierarchy. The IconNet enables the user to add concepts to the ontology, remove concepts, and apply higher-order user actions in a graphical environment using direct manipulation.

The iconology is in many ways different from WordNet, CYC or other ontologies. The same points (see again [20]), but now with the focus upon the similarities and dissimilarities with WordNet and CYC are:

- Generally: CYC is also a *general* ontology and WordNet a *general* lexical ontology. They contain much more concepts than the iconology: in the order of 10^5, respectively 10^4. CYC does have the representation language CYCL and is partially online. WordNet uses semantic networks and synsets (synonymy sets) and is published online. Both ontologies are used by many researchers;

- Design process: WordNet is built bottom-upwards and used a text corpus as the basis of their development process. CYC nor WordNet used automatic ontological knowledge acquisition. Both are constructed manually, like the iconology;

- Taxonomy: WordNet contains a single tree-like concept hierarchy for noun synsets with multiple inheritance and parallel hierarchies for adjectives and verbs. It uses hypernym. CYC uses the *distinctions approach*, what means that it does not have a fixed set of top-level categories. Concepts can be categorized using cross-classification, like in the iconology. At the other hand, CYC does have predefined top-level entities, like Tangible versus Intangible, and Individual versus Collection. CYC stores like the iconology all concepts in one ontology. WordNet does not contain a hierarchy of types. The iconology is sparse and tangled like CYC;

- Internal structure of concepts: CYC does have properties and roles assigned to concepts, WordNet does not. The relations in WordNet are binary what implies an internal structure of the same type as in the iconology (in which they are triadic). In CYC are roles defined upon concepts in the ontology itself. In the iconology concept roles are defined by external grammatical rules;

- Axioms: CYC does have axiomatic information incorporated in the language processing code. WordNet does have no axioms (like the iconology);

- Inference mechanism: CYC goes beyond first-order logic by reification (regarding a predicate as an object: property P is the opposite of property Q) and having contexts as first-class objects ("You cannot see someone's heart", is true except for surgery). WordNet contains no inference rules;

- Applications: CYC has an application in the form of CYC Natural Language System [CNL] that translates natural language texts into CYCL. It is also used in CYCCESS, a semantic information-retrieval system. WordNet is used for disambiguation in particular, for example in the Oingo and SimpliFind search engines [43].

The iconology is in some facets different, in some facets equal to WordNet and CYC. The reasons that the iconology is different originate from the visual (thus non-verbal) modality of the concepts it contains, its purpose (knowledge representation, no knowledge reasoning), and others that can be derived from subsection 3.1.3.7 and subsection 3.1.3.8. The following subsection higlights these points one more time.

7.2 Model Scope

The kind of solutions tackled with this model is limited. Next section (7.3) describes what this model offers. This section describes the limited scope of the model and framework. Remaining limitations, bugs and impracticalities are described in section 7.5. The model does have the follow characteristics:

- The iconology should be independent of verbal constructs, not only independent of words, but also independent of strings. The emphasis is therefore upon a *proper data representation*. No automation functionality like machine translations of texts or automatic knowledge aquisition from texts (both involving "comprehension") is intended. Automatic mapping

VilAug IconMessenger IconNet

between domain ontologies is not implemented. Machine processing is indirectly provided by defining a GraphML API for data transfer. The ontology is lightweight: knowledge reasoning exist only in the form of derving $A \rightarrow C$ from $A \rightarrow B$ and $B \rightarrow C$ if both relations denoted by the arrow are from exactly the same type (in the form of an icon.hasAncestor(ancestor, type) function);

- The "relational" icons between "normal" icons should not be restricted to the hypernymy, holonymy or antonymy type. There is an *extendible hierarchy* of relation types. Inference is thus not merely induction using hypernymy relations. There are also combinations possible, like that the relation hasColour implies the relation hasProperty or even isPerceptibleEntity. Such inferences are convenient for knowledge reasoning and not mere knowledge representation anymore;

- The iconology is intended for an application *context of an icon menu*. This might be considered an artificial limitation. But it restricts the amount of redundant information (for example it contains only generic to specific relations, not their reciprocals). And it also implies a directed acyclic graph constraint that prevents automatically cycles in which hypernymy relations contain themselves;

- The iconology uses a *standard format*, namely GraphML extended with attributes apt to the icon domain. This enables own model specifications like the applied external conformity relation that separates types from instances by distinct places intern and extern to the ontology.

- The iconology exist by the merit of the existence of *different specifications of conceptualizations*. It does not promote one unique way of labelling and modelling the yet unmodelled outerworld. There is therefore no fixed set of top-level entities, nor a fixed hierarchy of (top-level) relations, nor a fixed set of grammatical items or rules;

These are the most important characteristics of the model. They restrict the model, but on purpose. The mayor strenght of the iconology does not lay in a process of automatic content creation, nor enormous effort spent at defining the upper-level categories, but especially its visual modality.

7.3 Project Value

What kind of intrinsic value has this project, the (theoretical) model and the framework, added to the field of visual linguistics? There were already visual iconic languages available, like Lingua and VIL. In what way is the situation improved?

The theoretical model adds - perhaps - new information and viewpoints about characteristics of a visual language. It also has a new point of view in regard to ontology engineering in the way that words and verbal descriptions are considered of secondary importance. The way concepts, relations and grammatical relations are seen are influenced by their iconic nature. Concept, relational, grammatical, can be used across languages. The resulting iconology should make sense in the context of a hierarchical icon menu.

The way words and other verbal constructs are energetically banned from the model, is one of its most typical characteristics. The equal treatment of concepts, relations and grammatical concepts and relations is another very typical trait. The latter is a consequence of the former. All entities in the iconology are seen as icons (in popular terms: as icons, relaticons and grammicons). What exactly is the advantage of this non-discriminating approach? There are several reasons:

- *Reification of relations* makes it easier to use them as arguments in larger constructs or rules (icon type hierarchies, grammatical type hierarchies, grammatical rules);
- The fact that a data element *carries semantics* is taken most seriously, not its semantic type. Semantics is carried by icons in the iconology, hence everything that carries semantics is an iconic concept. That all elements are iconic concepts eases transformations from one type to another. No effort is needed to establish fuzzy concepts that linger on the border between two types.

The framework that contains the iconology. In what way is this framework important?

- The framework is a *demonstrator* of the iconology. It contains a graph as internal data object. It knows how to read and write a graph structure on the disk using the GraphML format;
- The framework adds a *querying version of inference*. There is no function that implies knowledge reasoning by simplifying data structures. There is a function that knows (in a search over $\rightsquigarrow$ types) to find C in an $A \rightsquigarrow B$ and $B \rightsquigarrow C$ data structure;
- The framework allows *several visual iconic languages* and grammars to be part of the iconology at once. Each language is represented by an icon that is related with icons that belong to the language. Each grammar is represented in the same way. Each grammar can also use its custom set of grammatical rules;
- The framework *visualizes* its data structure, the graph, with a GUI (graphical user interface) called IconNet. Connections between icons can in this way be found more easily than searching for identifying numbers across several text files. Grammar parsing can be visualized likewise;
- The IconNet environment does not merely visualize the graph, it also adds *direct manipulation* functionality so that the user can manipulate the data in this graphical environment. With graphical user actions - clicking an icon, dragging edges between icons - can vertices and relations be created;
- The IconNet environment is enhanced with *complex operations* (see subsection 3.2.3) such as “upgrade relation” and “interface concept”. These mutations are collections of smaller mutations that handle the creation or deletion of an individual vertex or edge. The complex mutations are implementations of graph rewriting techniques on a tiny scale.

An application that stores several visual languages, does not exist at all currently. Even less a visual language framework that uses a graphical environment to construct such visual languages.

7.4 Interface Design Analysis

Following versions of the VilAug framework should have the following focus in regard to design principles (see Shneiderman and Plaissant in [42]). Table 7.1 provides comments and encountered pecularities. Priority is difficult to assign quantitatively, so an intuitive number of order of *apparent neglection* is given in the priority column.

Table 7.1: Future Design Focus in regard to User Interface Design

principle	comments and encountered pecularities	priority	time
reversal	undo actions are not reversible	1	40
error prevention	it is made impossible to create relations that disobey the acyclic directed graph constraints, it is possible to cancel most actions (by right-clicking empty space or clicking cancel), all icons can be altered, also the ones in the “default” libraries	2	20
feedback	a hint panel is provided, the creation of a relation is entirely visualized by drawing two edges subsequently, there are many dialogs, no user manual	3	40
usability	no plasticity exists, apart from writing directly GraphML versus direct manipulation, no shortcuts, but the creation of entire relations and more sophisticated user actions (that can be seen as macros) are very novice and expert friendly	4	40
closure	only dialogs that are really necessary are shown, closure is automatically obtained by the mentioned macros; but seems not to have had special attention	5	4
control locus	the user initiates every action, there is however no macro to connect or add (tedious) sequences of icons	6	80
memory load	the icons are like folder in a treeview of windows explorer; grouping is not optimized to +/- 7 chunks	7	40
consistency	same dialogs are used, consistency is almost “too” complete: okay and cancel buttons do not make sense in every dialog (there are mere information dialogs)	8	8

The design is not done down to the level of positioning elements.

7.4.1 Content of Design Principles

Most attention deserves the reversal of actions. Currently nothing like an undo function exists, what refrains the user from actively exploring all possible actions. This does have much to do with reliability of the underlying data just as error prevention. The standard libraries should be made immutable and the linguist be able to perform changes, additions and deletions to parts of the topology that do not have very severe consequences.

After these issues that are about the integrity of the underlying data, the third and fourth topics of improvement are the provided feedback and plasticity. There is no user manual, there is no internet forum, there is no website that provides guidelines. At least some of these materials have to exist. Plasticity in the form of command-line instructions or short-cuts have also to be added.

Less important issues are closure, internal locus of control, the short-term memory load and consisteny. These seem to be handled quite well in the application and do not have the highest priority. The "create relation", "upgrade relation" user actions that are called macros overhere, turn out to obey the principle of closure semi-automatically.

7.4.2 Time involved in Implementation

It is difficult to indate how much time a given problem would take in terms of programming. The dialogs are easy to change, because they are derived from one or two parent dialogs. Changing their consistency takes no longer than one workday.

To add plasticity in the sense of a command line box, designing the commands, checking for typing mistakes, recognizing the commands, adding a module that recognizes key shortcuts, updating the model automatically on these commands will probably soon take a week.

Writing a user manual, creating an online support forum and supporting website will take that much too.

To check for real closure of all actions does not take more than half a day. Prevention of errors involves creating a manner in which the user is identified. The identification makes altering the corresponding part of the graph possible. This involves the addition of a user managing system that restricts the editing modules and takes at least half a week.

Adding undo functionality is conservatively planned upon one week, but it could be less. It entails the storage of each mutation event. Each mutation event should be decomposed in the smaller mutation events: create vertex, remove vertex, create edge, remove edge, change vertex, change edge. A class that stores these events in sets, and stores copies of old elements is needed.

Most time would take the creation of a macro for adding multiple icons at once or relating multiple icons at once. A way to select multiple icons has to be invented and adding mutiple edges to them. This is not explored at all until now and presumably takes much more than one week.

The memory load issue can be improved by considering the layout. There are a lot of layouts algorithms available. Their complexity can be compared with parsers. Implementing another layout algorithm corresponds with a lot of work and is estimated upon a week at least.

7.5 Framework Analysis

This section analysis the framework and tries to point out unintended (model and) framework limitations, impracticalities, not implemented functionality and bugs. In one of previous sections about the model comparison (7.1) are the limitations desired or foreseen. The topics in this section are not model limitations but implementation limitations. So, the results of this analysis will be adjusted by subsequent releases of the product.

7.5.1 General Model Functionality

The stakeholder involvement diagram (from figure 4.1 on page 73) can be used to indicate what has been implemented in this framework, and what has yet to be implemented. See figure 7.1 (gray ellipses are implemented).

The tasks and functionality around the icon ontology are especially the ones that are implemented entirely. To the functionality that is not implemented belong tools to aid the developer in creating the icons itself (see subsection 7.6.1). There are no semantic restrictions (see subsection 7.6.2).There are neither translation tools (see subsection 7.6.4).

7.5.2 Specific Model Functionality

Functions that could have been assigned to the implemented tasks and neither are implemented, are the following:

- A way to add icon properties like colour (without adding subicons with that particular properties in the iconology, see subsection 7.6.3);
- The mutation “irredudant concept removal” (subsection 3.2.3 on page 65 and further explained in subsection 3.2.3.2) is not implemented. The removal of a concept has currently no influence on its environment. Implementation difficulty is like that of the mutation “upgrade relation”;
- IconNet shows the popup-menu an option like “remove” above a vertex. These actions should be depicted as icons instead of words;
- There is no buddy list in the IconMessenger, it is only possible to communicate with one person, and the communication parameters can not be set in a user-friendly way (but by adding an IP address as parameter);
- The grammar has to be changed by altering grammar rules manually in XML files. No graphical user interface (using icons) is provided;
- A new language has to be added manually. The files (ontology, grammar, rules) have to be appended in proper GraphML format;
- The IconMessenger user is not able to choose an iconography, but icons of all iconographies are used;
- The IconMessenger user is not able to choose a specific domain of a certain language. Like all the icons that have to do with chess in a certain visual language and orthography;
- There is no general resource where icons are stored. Nor are there tools that synchronize this set with individual icon sets on user machines.

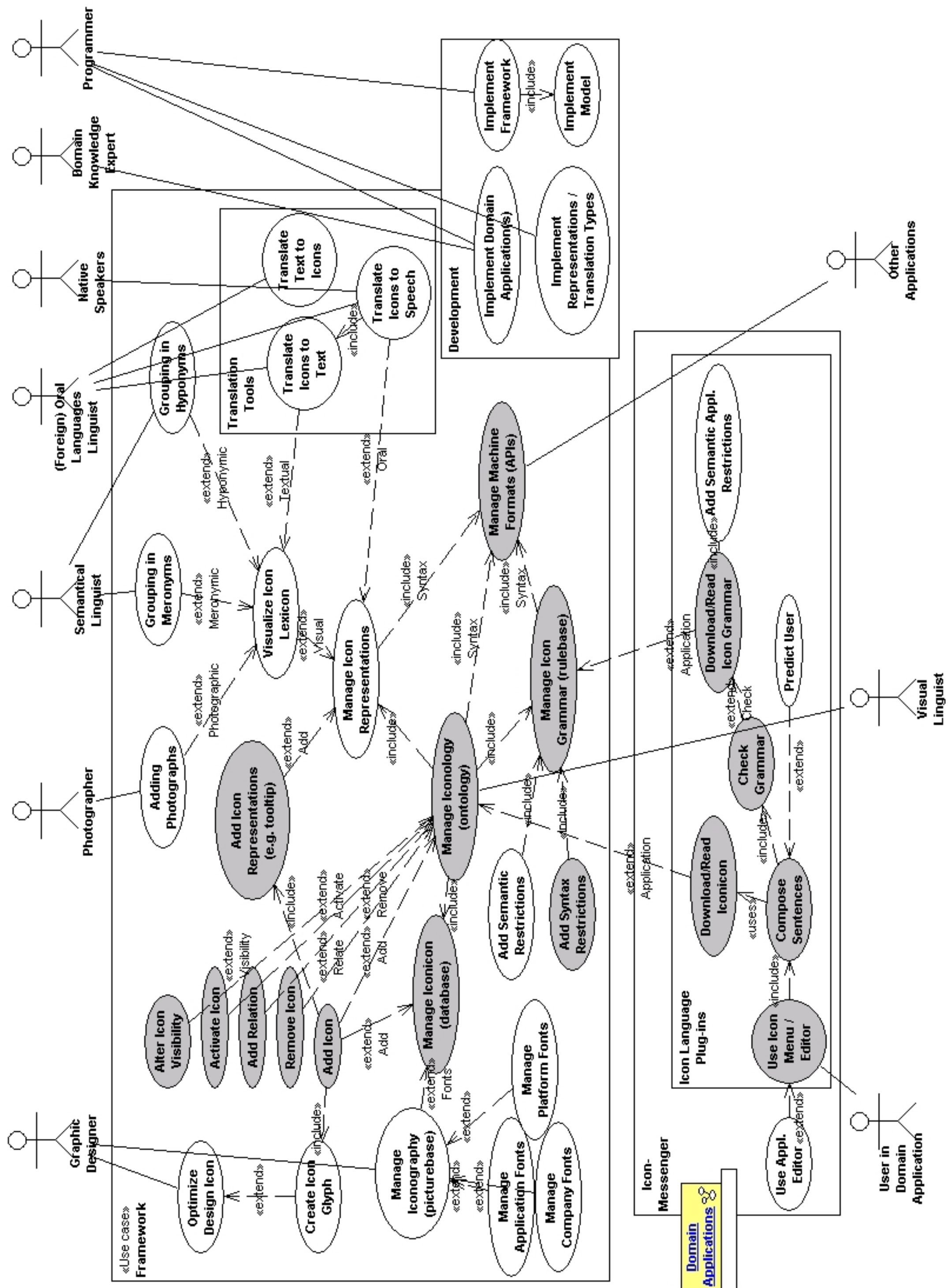


Figure 7.1: Stakeholder Involvement Diagram & Implemented Work

7.5.3 Specific Model Pecularities

The framework does have certain modules that can be bettered. The layout algorithm causes the vertices to overlap each other. No real overview can be obtained. Interface design topics are already discussed in section 7.4 and will be disregarded overhere. There remain relatively small issues like the disappearance of animated icons (those are possible!), when they become disabled. In general contains the application few indications about its abilities. More hint and help panels should be attached in general.

The framework is potential enough to store the visual languages VIL (subsection 2.3.2 on page 34) and Lingua (subsection 2.3.3 on page 36). Porting these languages to the framework has been successful. Although there are several restrictions. VilAug sustains the use of cascaded icons, but there is no way the user can enrich an icon with another icon. This will be called *adjective inflection*. This is like graphical inflection (see subsection 7.6.3) but involves the addition of an icon to another icon, not the change of icon properties itself. So, no icons can be attached as modifiers (or adjectives) to a main icon like in VIL. Another incomplete process is the assignment of grammatical classes to icons. Every icon or icon category should be assigned to the appropriate grammatical category, connecting for example FOOD with NOUN.

The issues discussed in this section should be handled in the next minor release of the software.

7.6 Recommendations

There are many practical issues and features that can be recommended because the framework is brand new in its kind. Some issues brought up in subsection 3.1.3.8 about model scope and limitations can complement the framework in a very useful way.

7.6.1 Orthography Tools

There are no tools that help the visual linguist, or rather the graphic designer (see figure 7.1) to design a picture set, also called orthography. It is possible to provide a tool that define graphic filters to provide an orthography in which icons have one particular style. It is a recommendation to create a tool to crawl the web in search for icons. Descriptions from surrounding icons can be used to find a new picture. When certain particular pictures are find, descriptions can be given and they can be embedded in the ontology. This is a recommended procedure, because it inverts the common process in which the visual linguist needs an icon for a certain word or verbal phrase. Now its starts with a found icon and the visual linguist has to invent text to describe that particular iconic concept. This is - again - against verbocentrism (assumption 14) a central theme in this model.

7.6.2 Semantic Restrictions

The icon menu that is based upon the iconology is gouverned by grammatical rules that disable items on each menu level. These grammatical rules define which icon can occur with which other icon in one sentence. The grammatical

parser indicates which ancestors (see subsection 4.8) are required for a new icon. There is no difference between searching for a grammatical icon as ancestor and a normal icon as ancestor. There is however one important issue. It should also be possible to interpret the rules differently. Currently (no formal term) *allowance parsing* is used. Combinations of icons formed by the rules are *allowed*. To formulate all kind of semantic combinations in this way, would be very difficult. It is better to start with *denial parsing*. Combinations of icons formed by the rules are in that case *denied*. Semantic restrictions like ANIMALS SPEAK can be applied in this case.

7.6.3 Graphical Inflection

The icons can be adapted directly by adding a second icon to a concept, or by adjusting its graphical properties. Changing its graphical properties in such a way will be called *graphical inflection*. It can be the case that adjectives in a verbal language seems to be represented most appropriate by inflection of an icon. And not by inventing a separate icon for such adjective. An inflected icon can be seen as a variant upon an icon, or a particular subtype. They can be seen as refined versions of a particular concept. For example, a WHITE GLASS and a GREEN GLASS are both subtypes of GLASS, as well as EMPTY GLASS and FULL GLASS. The concepts WHITE, GREEN, EMPTY and FULL do not have to be visualized as separate icons.

It is possible to apply graphical inflection using another (popup) menu. Features that can be added by inflection should be offered via that menu. In the menu all kinds of adjectives are shown as seperate icons. Merging an adjective with its main icon can be done by standard procedures (colouring, resizing). Because adjectives are in this way just more detailed iconic concepts it should be possible to turn them off (and on). Both, the specific treatment of grammatical inflection and automation in regard to merging procedures, can be implemented in framework sequences.

7.6.4 Translation Tools

There are no translation tools added to the model. Maybe translation would have seemed a natural add-on to implement first. However, to create verbal phrases from iconic message is a hard problem. The system has to be able to write grammatical correct English and should know how to conjugate verbs, to inflect nouns, where to add determiners, apply correct word order, etcetera. By the way, this sentence reflects a certain bias. A translation tool that translates to another language like German, Mandarin or Portuguese would be fine too. There is no *the* translation tool. Every translation module can be seen as an addition to explain the meaning of the icons in a certain form. This can be in the form of gestural signs (or symbols, see SignWriting in subsection) or photographs, or speech, or by ordering them together in groups. The last recommendation can be called *translation by grouping*. Words tend to be explained by running text instead of a set of keywords. Icons are perhaps better explained in the latter way.

7.6.5 Implementation Improvements

The programmer can also improve certain matters. User management can be made more sophisticated. It that way it will be easier to connect data changes with users, and user responsability. A good enhancement of the current logging in and out procedures, is a SecurityModule. A default setting disallows the user from editing any language. Or changing pictures, grammar or grammar rules. An authorized will be able to change a language. It is possible to use passwords in this environment that are pictures of these users encrypted with a hidden graphical key.

The application is written for version 1.4 of the Java Runtime Environment [JRE] and is therefore typesafe. This means that if the type of an object is used incorrectly, at least a ClassCastException will be thrown. A new technique is added in version 5.0 of the JRE. This technique is called *generics*. Generics provide *static type checking*. This means that there will be no runtime exceptions anymore like described above. The types can be checked during *design time*. It is also recommended to apply *generified* design patterns like described by Spritzler [44]. Further implementation recommendations can be found in the source code.

7.6.6 Usability Assessment

The tools IconNet and IconMessenger may profit from a usability analysis. In this chapter an analysis from the viewpoint of the developer is given. This thesis work had as goal to provide the visual linguist with a model that fits the needs of the visual modality, the iconology. It also provides the linguist with an environment to manage visual languages that make use of the iconology. The functionality to handle such languages is provided. However, the focus of this thesis did not lay upon usability. An experiment to test the usability of the tools with actual users is recommended.

Chapter 8

Conclusion

Visual languages contain icons instead of text. The transmission of icons instead of text may offer advantages in regard to ease of communication. How should such a visual language be represented digitally? What are salient characteristics of this model? How to design a software framework that uses this model? What exactly is implemented in this research assignment?

Section 8.1 describes main characteristics of the system. Section 8.2 compares the model with existing work. Section 8.3 finishes with the results.

8.1 System

The discipline of visual linguistics regards visual languages that use icons to communicate. A new ontological model, called an iconology, is developed. It *explores the visual modality* in several dimensions. Verbocentrism, the tendency to use words for mental concepts, is diminished with several techniques. The Iconology contains references to the icons themselves, no verbal descriptions. Relations between icons are icons themselves, no verbal descriptions. Types are icons. Grammatical concepts are icons. Even grammatical rules are not limited to the familiar "verb", "noun", "adjective" classes, but may impose restrictions upon whole other types of grammatical classes.

A new software model, labelled VilAug, is developed. It *separates visual language creation from programming*. The visual linguist is able to create and alter visual languages in the sense of adding icons, deleting icons, adding relations, adding grammatical concepts, etcetera without the need for programming (using the GraphML syntax). The VilAug framework also stores *several visual languages* (and grammars) in parallel. Applications that use this framework are enriched with a range of visual languages, instead of only one language.

A novel graphical environment, called IconNet, is developed. It visualizes the iconology, and above that, it enables the visual linguist to use (direct) *graphical manipulation*. The actions mentioned above can be performed in a graphical user interface without the need to know the GraphML syntax.

Another tool, the IconMessenger, is provided. It is an *instant messenger*, containing an icon menu representation of the Iconology. The user is able to navigate iteratively through the menu items to compose a message.

8.2 Comparison

The iconology is an adapted version of a *conceptual graph*. A conceptual graph has both concepts, and (reified) relations tied to its vertices. The difference being the existence of *merged hierarchies.* The concept hierarchy is merged with the type hierarchy and relation hierarchy. The iconology is represented by a GraphML syntax with several icon specific keys defined as GraphML attributes. The grammar is represented in a custom XML format reflecting BNF syntax. The grammar parser recognizes context-free grammars.

The difference between this system and previous work is the existence of several visual languages in parallel in one system. Previous applications did not allow the user to switch to another visual language. Nor did they allow the user to add an entire new visual language. Another new feature is the *discoupling* between iconicon (lexicon of icons) and grammicon (lexicon of grammatical items). Several types of grammars are allowed for one language. With also their own set of grammar rules. It is possible to use different grammar classifications, like one using cases like “actor”, “instrument”, “locative” or one using “noun”, “verb”, “adverb”. Two types of grammar are stored. A “frequency grammar” comparable to “case grammar” and “form change grammar” or “inflection grammar” comparable to ordinary grammar. Previous applications did neither provide a graphical user interface to manipulate and construct visual languages. There is at least one instant messenger - Lingua, one of the predecessors of VilAug - that uses only icons to communicate.

The difference between CYC and WordNet is the absence of a fixed set of *upper-level categories* in the ontology. A difference with WordNet is the absense of a fixed set of *relation types*. Another difference is the priority given to the graphical shell around the iconology, IconNet. Visualization of the iconology is no add-on. The user is able to manipulate the iconology by graphical manipulation.

8.3 Results

The ontological model, the iconology, is used by the implemented framework VilAug. The latter functions as a demonstrator of the iconology. It shows that mechanisms like relating, interfacing and inheriting work. The two tools IconNet and IconMessenger can be seen as demonstrators of the framework VilAug. The ontology manipulations in IconNet result in changes in the icon menu in the IconMessenger. No usability assessment - other than a walk-through by the developer - has been performed.

The framework VilAug contains the two languages VIL (Visual Inter Lingua by Leemans) and Lingua (by Fitrianie) of which the feasibility was examined in their own research. The languages VilAug and Lingua function as the third type of demonstrators involved in this project. They demonstrate that the framework and functionality is in general rich enough too store these languages. Their (ordered and unordered) grammars are processed according to one grammar resembling ordinary grammar and one resembling case grammar. The adjective inflection (modifying an icon by cascading icons) in VIL is not preserved.

The software products of this thesis work can be downloaded from `http://sourceforge.net/projects/vilaug` or `http://vilaug.sourceforge.net`.

Figure 8.1 shows a screenshot of the IconNet and IconMessenger tools.

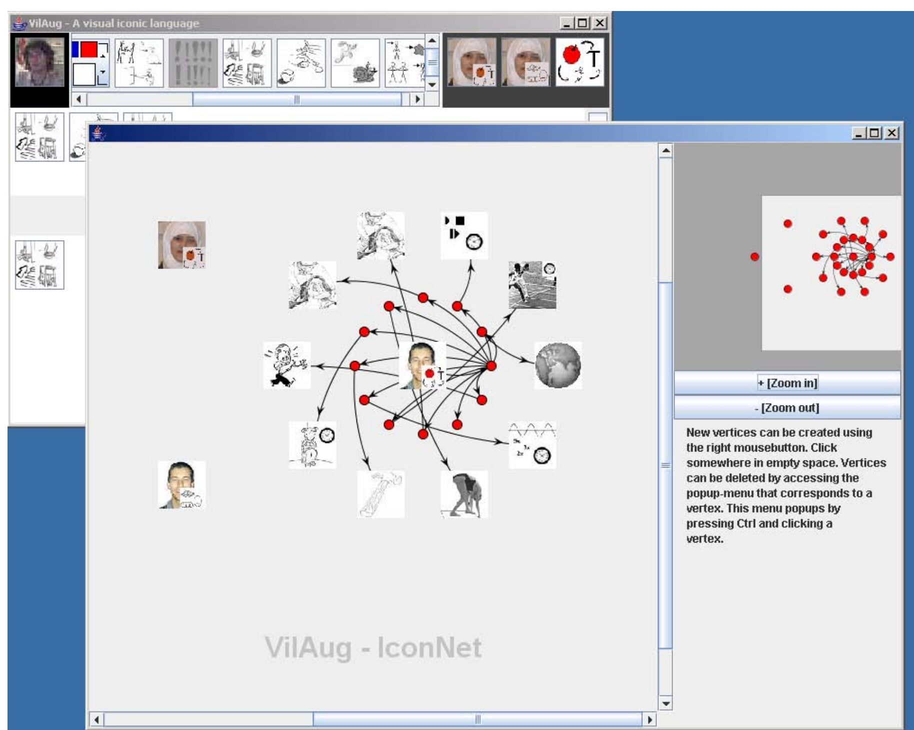


Figure 8.1: IconNet & IconMessenger

8.4 Future Directions

The developed model and framework represent knowledge in a way that is independent of verbal language. This is valuable because an ontology should contain concepts rather than words or phrases (or icons). The latter being a specific representation of a concept. It would be in the same spirit to develop an ontology that stores signs for sign languages. The corresponding framework would provide menu-options, etcetera in terms of signs rather than words (or icons).

Before people actually start to use the tools compatible and delivered with this framework - the IconMessenger and IconNet - will a usability assessment be necessary. Analysis of the visual languages is the responsability of the linguists themselves. The layout of the IconMessenger can be more like existing instant messengers, with buddies and facilities as sending files, setting presence state, etcetera. The way icons are entered can be optimized for the icon menu. The IconNet environment may facilitate the user with macros, several layouts, icon search methods, etcetera.

From a helicopter view the model and the tools are among the first that try to exploit the visual modality again. Was it the lack of writing speed that gave alphabets so much benefit over pictorial writing systems? With this era of digital computers the emphasis may shift again! An entire new discipline waits to be researched.

VilAug IconMessenger IconNet

Appendix A

Iconicity - More Dimensions

This appendix contains a specification of the visualization proposed by Cohn in figure 2.1.

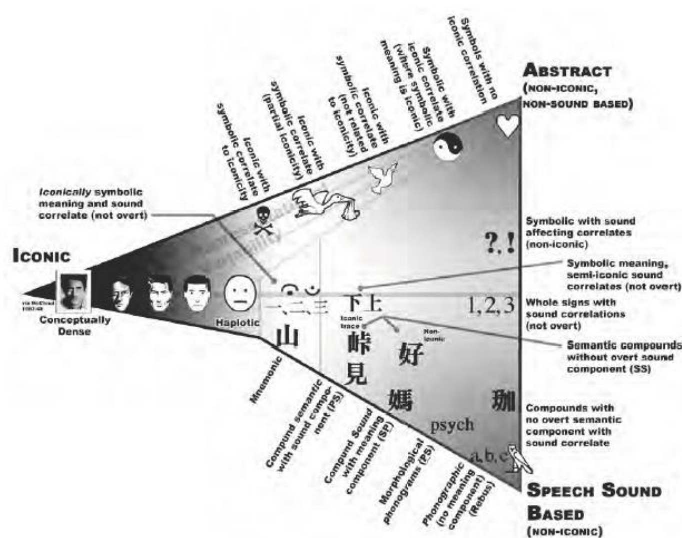


Figure A.1: Cohn's CMSG Diagram

Figure A.1 shows several dimensions of iconicity. From iconic to abstract, as well as from iconic to speech sound based. A third dimension can be imagined that adds another kind of diagrammatic iconicity that adds gestures of sign languages to this picture. For further information about for example the form of this triangle I refer to Cohn's documents. To return to the text about iconicity, return to subsection 2.1.2.2 on page 19.

Appendix B

Categorization in VIL & Lingua

This appendix contains a few categorization details about the sustained languages, VIL and Lingua. The tables here serve as appendix to section 2.3 on page 32.

B.1 Modifiers in Fillmore's Case Grammar

In VIL several elements are taken from Fillmore's Case Grammar and Schank's Conceptual Dependency Theory. From the first a lot of its grammar has been taken. The latter delivered mainly clues how to divide the conceptual field of verbs. One of the elements in case grammar are the modifiers. The user has to chosen first the "Mood" (declarative, interrogative, imperative) from Fillmore's types of modality. The other modifiers Fillmore defined are listed in table B.1.

Table B.1: Modifiers in Fillmore's Case Grammar, used in VIL

modality	example
Tense	present, past, future
Aspect	perfect, imperfect
Form	simple, emphatic, progressive
Mood	declarative, interrogative, imperative
Essence	positive, negative, indeterminate
Modal	may, can, must
Manner	adverbial (diligently)
Time	adverbial (at that moment)

You can find how they are more or less embedded in the grammar in VIL in subsection 2.3.2 on page 34.

B.2 Primitive ACTs in Schank's Conceptual Dependency Theory

The semantic consituents that form the basis of each verb are called ACTs by Schank. He declares such primitive ACTs in B.2.

Table B.2: Primitive ACTs in Schank's Conceptual Dependency Theory

ACT	description	example
ATRANS	Transfer of an abstract relationship	give
PTRANS	Transfer of the physical location of an object	go
PROPEL	Application of physical force to an object	push
MOVE	Movement of a body part by its owner	kick
GRASP	Grasping of an object by an actor	clutch
INGEST	Ingestion of an object by an animal	eat
EXPEL	Expulsion of something from the body of an animal	cry
MTRANS	Transfer of mental information	tell
MBUILD	Building new information out of old	decide
SPEAK	Production of sounds	say
ATTEND	Focusing of a sense organ toward a stimulus	listen

Also these Schank's list of primitive ACTs are related to VIL in subsection 2.3.2 on page 34.

B.3 Categorization of Verbs in VIL

The first division in subclasses of the main verbs in VIL is depicted in table B.3.

Table B.3: Categorization of Verbs in VIL

nr.	category	subclasses
1	Mental	Psychological State, Attend, ...
2	Alienable Possession	Atrans, Hold & Keep
3	Physical Location & Motion	Ptrans, Propel, Sending & Carrying, Lodge, ...
4	Existence	Create, Exist, Destroy, ...
5	Identificational Appoint	Appoint, Weather, Measuer, Identity, ...
6	Involuntary	by Non-humans, by Humans, Sound Emission, ...
7	Voluntary	Move, Touch & Poke, Ingest, Expel, Sports, ...

To return to the theory chapter about VIL, see subsection 2.3.2 on page 34.

B.4 Categorization of Nouns in VIL

The first division in subclasses of the main nouns in VIL is depicted in table B.4. Originally the most important nouns are kept separated from less important

nodes in two different tables in an Microsoft Access database. This is by the way also the case for VIL's verbs.

Table B.4: Categorization of Nouns in VIL

nr.	category	subclasses
1	Physical World	Universe & Space, Geography, Geology
2	Beliefs, Customs and Society	Law & order, Religion, Customs, Money, ...
3	Arts & Entertainment	Theater, Music & Dance, Painting & Sculpture, ...
4	Sports	Athletics, Display, Target, Court, Team, ...
5	Communication	Language, Mail, Computer, Electronic
6	Science & Technology	Physics & Chemistry, Time & Measurement, Machines, ...
7	Transportation	by Road, by Rail, by Sea, by Air, through Space

To return to the theory chapter about VIL, see subsection 2.3.2 on page 34.

B.5 Categorization of Nouns in Lingua

The first division in subclasses of the main nouns in Lingua is depicted in table B.5. Not all categories have subclasses, so some entries remain empty. There are subitems until level three. So, the items in "subclasses" will not be refined very much further. This is of course not very strange because it is a perfect example of a domain ontology.

Table B.5: Categorization of Nouns in Lingua

nr.	category	subclasses
1	People	Pronouns, Jobs
2	Verbs	by Eyes, by Ears, by Mouth, by Hand, by Thought, ...
3	In the Restaurant	Ordering, Beverage, Bread, Desert, Meal, Meat
4	In the Market	Fish, Fruit, Vegetables
5	In the Shop	Purchasing, Accessoires, Electronics
6	In the Hotel	
7	Money	
8	Time	
9	Extras	Number, Sign, Colour, Perception
10	Adjectives	Adjective, Emotion
11	The World	Country, Famous Places
12	In the City	City, Landscape
13	Transportation	Vehicles, Places, Direction

To return to the theory chapter about Lingua, see subsection 2.3.3 on page 36.

Even more details can be found in the documentation of the developers of VIL and Lingua [2].

Appendix C

Porting to VilAug

C.1 Porting from VIL and Lingua to VilAug

This appendix will briefly review the code that had to be written to use the data from Lingua and VIL and incorporate it in the framework VilAug. VIL was written in *Visual Basic* and did store its icons (as blobs) and its hierarchy (as tables) in a *Microsoft Access Database*. The database was stored in a way so that an iconic concept could be listed more than once in a table. Each time with a different parent. SPACECRAFTS did have for example the parent TRANSPORTATION and the parent UNIVERSE. This form of lattice is preferred when events of cross-classification are rather sparse. The implemented iconology however used cross-classification on a much bigger scale. This is another reason for the graph representation (see subsection 3.1.3.7 on page 51).

To translate the VIL database to VilAug's GraphML files, the tables had to be browsed upon "icon - icon parent" pairs. The output results were written in GraphML files. Because of its perfect database handling, *Delphi* was used to perform this operation. The icons were also imbedded in the database, but it was impossible to retrieve them all. This because of a bug regarding the OLE Container Control that is not designed to work with the ADO Data Control. Fortunately, Leemans send a copy of all the icons used in VIL.

The application Lingua is written in Java. The icons are stored in a directory instead of a database. the relationships between them are stored in a file, labelled "family.xml". That file contains a tree structure with grammatical items ordered in levels. Cross-classification is impossible due to the tree structure. Lingua uses a *custom XML* notation. The file in Lingua that stored the relationships between icons, had to be transferred to the GraphML format. For that purpose an *XSLT stylesheet* was written (called lingua2graphml.xsl) and some secondary stylesheets to order the graph elements and add whitespace.

The functionality of these both helper tools are considered from too temporarily nature to be incorporated in the report itself, hence their description overhere.

Appendix D

Picture Credit

These pictures are not my own and deserve credit. Credit to the picture or pictures in:

- Figure 2.1, A.1: Cohn [8];
- Figure 2.2: Gasser [9] (actually from presentation);
- Figure 2.7: Chang [32];
- Figure 2.4: WorldPeace, the flag (from `http://www.worldpeace.no/`), Bump, the painting (from `http://www.cwrl.utexas.edu/~bump/`);
- Figure 2.5, 2.6: Sowa (from `http://www.jfsowa.com/cg/cgexampw.htm`);
- Figure 2.8: Leemans (own screenshot);
- Figure 2.9: Fitrianie (own screenshot);
- Figure 4.5: Kamps (from `http://staff.science.uva.nl/~kamps/wordnet/`) and Treebolic (from `http://treebolic.sourceforge.net/`);
- Figure 4.6: CharGer (from `http://charger.sourceforge.net`).

Figures containing graphs are made in GraphViz (GraphViz Homepage `http://www.graphviz.org/`). Figures containing UML are created in ModelMaker (Borland Delphi). The icons in the application stem from the original applications and have the same copyrights.

VilAug
IconMessenger IconNet

Appendix E

Class & Interface Hierarchy

This appendix contains a hierarchy of all classes and interfaces from the created framework. The ones created by myself are bold.

Package Hierarchies: jung.ext, jung.ext.actions, jung.ext.dag, jung.ext.elements, jung.ext.graphml, jung.ext.icon, jung.ext.mutations, jung.ext.mutations.mutators, jung.ext.predicates, jung.ext.predicates.edge, jung.ext.predicates.edge.impl, jung.ext.predicates.general, jung.ext.predicates.vertex, jung.ext.predicates-.vertex.impl, jung.ext.registry, jung.ext.utils, jung.refact, thd, vilaug, vilaug.components, vilaug.exceptions, vilaug.grammar, vilaug.icon, vilaug.io, vilaug.peirce, vilaug.peirce.mutators

Class Hierarchy

The class hierarchy starts with the Java object.

- class AbstractAction (implements Action, Cloneable, Serializable)
  - class **AbstractMutationAction**
    - class **CreateEdgeAction**
      - class **CreateDirectedEdgeAction**
      - class **CreateUndirectedEdgeAction**
    - class **CreateRelationAction**
    - class **CreateVertexAction**
    - class **RemoveEdgeAction**
    - class **RemoveVertexAction**
- class **AbstractGraphMLElement** (implements GraphMLElement)
  - class **DefaultGraphMLElement**
    - class **LabellingGraphMLElement**
      - class **IconGraphMLElement**
- class AbstractGraphMousePlugin (implements GraphMousePlugin)
  - class AbstractPopupGraphMousePlugin (implements MouseListener)

        - class **EditingPopupGraphMousePlugin** (implements ActionFactoryDecorator, VisualizationControllerDecorator)
            - class **EditingPopupIconGraphMousePlugin**
    - class EditingGraphMousePlugin (implements MouseListener, MouseMotionListener)
        - class **EditingGraphMousePlugin** (implements BasicInterface, MutationFactoryDecorator, VisualizationControllerDecorator)
            - class **EditingPeirceGraphMousePlugin**
    - class PickingGraphMousePlugin (implements MouseListener, MouseMotionListener)
        - class **PickingGraphMousePlugin**
            - class **CtrlPickingGraphMousePlugin**

- class AbstractLayout (implements ChangeEventSupport, Layout)
    - class **AbstractLayout**
        - class **MutationLayout** (implements LayoutMutable, SettableVertexLocationFunction)
            - class **StaticLayout**
                - class **EccentricLayout**
- class **AbstractMutator** (implements GraphMutationListener, MutationFactoryDecorator)
    - class **AbstractEdgeMutator**
        - class **CreateEdgeMutator**
        - class **RemoveEdgeMutator**
    - class **AbstractPeirceMutator**
        - class **ActivateIconMutator**
        - class **CreateRelationMutator**
    - class **AbstractVertexMutator**
        - class **AbstractIconVertexMutator**
            - class **AddVisiblenessMutator**
            - class **OpenOrCloseVertexMutator**
            - class **OpenVertexMutator**
        - class **CreateVertexMutator**
        - class **RemoveVertexMutator**
    - class **AbstractVertexPairMutator**
        - class **ChangeVertexMutator**
- class AbstractRenderer (implements Renderer)
    - class PluggableRenderer (implements HasShapeFunctions, PickedInfo)

    - ▷ class **PluggableRenderer** (implements HasShapeFunctions, PickedInfo)

- class **AbstractType** (implements BasicInterface)
  - ◦ class **DialogMode**
    - ▷ class **DefaultDialogMode**
    - ▷ class **PeirceDialogMode**
  - ◦ class **ElementProperty**
  - ◦ class **GraphMLAttributeType**
  - ◦ class **GraphMLElementType**
  - ◦ class **GraphMLReadingMode**
  - ◦ class **GraphMLValueType**
    - ▷ class **IconValue**
  - ◦ class **IconContent**
  - ◦ class **InitializationMode**
  - ◦ class **KeyType**
  - ◦ class **MouseMode**
  - ◦ class **MouseModeName**
  - ◦ class **MutatorMode**
    - ▷ class **PeirceMutatorMode**
  - ◦ class **MutatorModeName**
  - ◦ class **NormalForm**
  - ◦ class **PluginMode**
    - ▷ class **DefaultPluginMode** (implements Serializable)
  - ◦ class **PluginName**
  - ◦ class **PredicateMode**
  - ◦ class **PredicateMutation**
  - ◦ class **PredicateType**
  - ◦ class **VertexType**
    - ▷ class **IconVertexType**
- class **ActionLogger**
- class **ArchetypeLayout** (implements ChangeEventSupport, Layout)
- class **BasicUtils**
- class **CheckMarkIconListener** (implements ItemListener)
- class **Completer**
- class Component (implements ImageObserver, MenuContainer, Serializable)

VilAug IconMessenger IconNet

- class Container ⟹ class JComponent (implements Serializable)
  - class AbstractButton (implements ItemSelectable, SwingConstants)
    - class JButton (implements Accessible)
      - class **IconButton**
  - class JPanel (implements Accessible)
    - class **DialogPanel**
      - class **DefaultIconDialogPanel**
        - class **LoginPanel**
        - class **LogoutPanel**
    - class **GraphZoomScrollPane**
    - class **IconInputPanel**
    - class **IconPanel**
    - class VisualizationViewer (implements ChangeEventSupport, ChangeListener, HasGraphLayout, LayoutTransformer, Transformer, ViewTransformer)
      - class **SatelliteVisualizationViewer**
      - class **VisualizationViewer**
- class Panel (implements Accessible)
  - class Applet
    - class JApplet (implements Accessible, RootPaneContainer)
      - class **VilAugMain**
- class Window (implements Accessible)
  - class Dialog
    - class JDialog (implements Accessible, RootPaneContainer, WindowConstants)
      - class **IconDialog**
      - class **IconInputDialog**

- class ComponentAdapter (implements ComponentListener)
  - class **GraphZoomScrollPane.ResizeListener**
- class **ComponentFactory**
- class **CustomRuleElementSet**
  - class **RuleElementSet**
- class **Data**
- class **DataController** (implements BasicInterface, ElementCheckerDecorator, ElementFactoryDecorator, RepertoireDecorator)
- class **Debugger**
- class **DefaultActionFactory** (implements ActionFactory)

- class **DefaultElementChecker** (implements ElementChecker)
  - class **DAGElementChecker**
    - class **IconElementChecker**
      - class **DefaultPeirceElementChecker** (implements PeirceElementChecker)
- class **DefaultElementFactory** (implements ElementFactory)
  - class **DAGElementFactory**
    - class **IconElementFactory** (implements GraphDecorator)
- class **DefaultGraphMutation** (implements GraphMutation)
- class DefaultHandler (implements ContentHandler, DTDHandler, EntityResolver, ErrorHandler)
  - class GraphMLFileHandler
    - class **GraphMLFileReader** (implements RepertoireDecorator)
- class **DefaultMutationFactory** (implements MutationFactory)
  - class **DefaultPeirceMutationFactory** (implements PeirceMutationFactory)
- class DefaultSettableVertexLocationFunction (implements SettableVertexLocationFunction)
  - class **DAGCircleSettableVertexLocationFunction**
- class **DefaultVerboCentro** (implements VerboCentro)
  - class **ApplicationVerboCentro**
  - class **GrammarVerboCentro**
- class DefaultVertexIconFunction (implements VertexIconFunction)
  - class VertexIconAndShapeFunction (implements VertexShapeFunction)
    - class **IconVertexIconAndShapeFunction**
- class DefaultVisualizationModel (implements ChangeEventSupport, VisualizationModel)
  - class **MutationVisualizationModel** (implements BasicInterface)
    - class **PeirceVisualizationModel**
- class **Dot**
- class **DummyUserDataContainer** (implements UserDataContainer)

  - class **DummyElement** (implements Element)

- class **EccentricData**
- class **EccentricData.Gender**
- class **EccentricLayout.EccentricVertexData**
- class **EdgeStringLabeller** (implements EdgeStringer)
- class **EditingGraphMousePlugin.ArrowShapeFunction** (implements EditingGraphMousePluginShapeFunction)
- class **EditingGraphMousePlugin.EdgeShapeFunction** (implements EditingGraphMousePlugin.ShapeFunction)
- class **EditingGraphMousePlugin.NodeShapeFunction** (implements EditingGraphMousePlugin.ShapeFunction)
- class **EditingGraphMousePlugin.ShapePaintable** (implements VisualizationViewer.-Paintable)
- class **EditingMode**
- class **ElementStack**
- class **EmptyElementDialog** (implements ElementDialog, GraphDecorator)
  - class **IconElementDialog** (implements IconGraphDecorator)
    - class **DefaultPeirceElementDialog** (implements PeirceElement-Dialog)
- class EventObject (implements Serializable)
  - class **GraphMutationEvent**
- class GPredicate (implements Predicate)
  - class **ElementPredicate** (implements BasicInterface)
    - class **ContainsUserDataKeyPredicate** (implements KeyDecorator)
      - class **StorablePredicate**
      - class **UnderConstructionPredicate**
    - class **PreconfiguredPredicate** ⟹ **MultiplePredicates** ⟹ class **DefaultPredicate** (implements RememberLast)
      - class **DefaultEdgePredicate** (implements EdgePredicate)
        - class **EveryEdgePredicate**
          - class **IncidentEdgePredicate**
          - class **PeirceEdgePredicate**
      - class **DefaultVertexPredicate** (implements VertexPredicate)

VilAug IconMessenger IconNet

        - class **ChildPredicate**
        - class **TillChildPredicate**
        - class **EveryVertexPredicate**
        - class **NoneVertexPredicate**
        - class **ParentPredicate**
            - class **TillParentPredicate**
        - class **PeirceVertexPredicate** (implements BrowsingDecorator)
        - class **ThisPredicate**
            - class **ExcludesPredicate**
            - class **IncludesPredicate**

- class **GrammarManager** (implements GrammarModule)
- class **GrammarParser**
- class **GrammarWeavers**
- class **GraphMLFile** (implements GraphFile)
- class **GraphMLFileWriter**
    - class **IconGraphMLFileWriter**
- class **GraphUtils**
- class **IconGraphUtils**
- class **IconManager** (implements IconModule)
- class **IconNetPanel** (implements BasicInterface)
- class **IconProperties**
- class **IconVertexPredicate** (implements Predicate)
- class ImageIcon (implements Accessible, Icon, Serializable)
    - class **DefaultImageIcon** (implements UberImageIcon)
        - class **DefaultIcon** (implements VertexIcon)
    - class **ScalableIcon**
        - class **LayeredIcon**
- class **Initializer**
- class **InterIconPredicate** (implements Predicate)
- class **IOInfoParser**
- class **IOManager**
- class **Key**

- class **LoginFactory** (implements PanelFactory)
- class **LogoutFactory** (implements PanelFactory)
- class **MouseModes** (implements BasicInterface)
- class **MutationType**
    - class **PeirceMutationType**
- class **MutatorModes** (implements BasicInterface)
- class **Pair** (implements BasicInterface)
- class **PeircePanelFactory** (implements PanelFactory)
- class **PeirceRelation**
- class **Perms**
- class **PickingGraphMousePlugin.PickingMode**
    - class **CtrlPickingGraphMousePlugin.CtrlPickingMode**
- class PluggableGraphMouse (implements VisualizationViewer.GraphMouse)
    - class EditingModalGraphMouse (implements ItemSelectable, ModalGraphMouse)
        - class **EditingModalGraphMouse**
- class **PredicateUtils**
- class **Predictor**
- class **RandomVertexLocationDecorator** (implements VertexLocationFunction)
- class **Replacement**
- class **Rule**
- class **RuleNode**
- class **Rules**
- class **SaussureRelation**
- class **Scanner**
- class ShapePickSupport (implements PickSupport)
    - class **ShapePickSupport**
- class **SingletonRegistry**
- class **StandardGraphMLRepertoire** (implements GraphMLRepertoire)
    - class **IconGraphMLRepertoire**

- class Throwable (implements Serializable)
  - class Exception
    - class **EdgeStringLabeller.UniqueLabelException**
    - class **GraphMLException**
    - class **ReplacementException**
    - class RuntimeException
      - class FatalException
        - class **FatalException**
          - class **InitException**
          - class **MutationException**
          - class **TypeException**
      - class **TranslationException**
- class ToolTipFunctionAdapter (implements ToolTipFunction)
  - class **FilteringTooltipFunction**
- class **Triple**
- class UserDataDelegate (implements Cloneable, UserDataContainer)
  - class AbstractArchetypeGraph (implements ArchetypeGraph, Cloneable)
    - class AbstractSparseGraph (implements Cloneable, Graph)
      - class SparseGraph
        - class **DefaultDirectedGraph** (implements DirectedGraph)
          - class **DAG**
          - class **GrammarGraph**
        - class DirectedSparseGraph (implements DirectedGraph)
          - class **IconGraph** (implements IconConverter)
  - class AbstractElement (implements Cloneable, Element)
    - class AbstractArchetypeEdge (implements ArchetypeEdge) ⟹ AbstractSparseEdge (implements Edge)
      - class DirectedSparseEdge (implements DirectedEdge)
        - class **GrammarEdge**
    - class AbstractArchetypeVertex (implements ArchetypeVertex) ⟹ AbstractSparseVertex (implements Cloneable, Vertex) ⟹class SimpleSparseVertex ⟹class SparseVertex
      - class **DefaultFamilyVertex** (implements FamilyVertex)
        - class **ReplaceableFamilyVertex** (implements FamilyVertex)
          - class **LimitFamilyVertex** ⟹ class **DefaultIconVertex** (implements IconConverterDecorator, IconVertex)
          - class **GrammarVertex**

VilAug IconMessenger IconNet

- class **VertexPair** (implements BasicInterface)
- class **VilAugViewer**
- class **VisualizationController** (implements BasicInterface, MutatorHandler, VisualizationViewerDecorator)
- class **VisualizationRegistry** (implements ActionFactoryDecorator, BasicInterface, ElementCheckerDecorator, ElementDialogDecorator, ElementFactoryDecorator, GraphDecorator, MutationFactoryDecorator, PanelFactoryDecorator, VisualizationControllerDecorator, VisualizationViewerDecorator)

Interface Hierarchy

- interface **ActionFactoryDecorator**
- interface **BasicInterface**
  - interface **Container**
    - interface **ElementDialog** (also extends PanelFactoryDecorator)
      - interface **PeirceElementDialog**
    - interface **ElementFactory** (also extends ElementDialogDecorator)
    - interface **GraphMutation**
    - interface **PeirceElementDialog**
  - interface **ElementChecker**
    - interface **PeirceElementChecker**
  - interface **ElementDialog** (also extends Container, PanelFactoryDecorator)
    - interface **PeirceElementDialog**
  - interface **ElementFactory** (also extends Container, ElementDialogDecorator)
  - interface **FamilyVertex** (also extends Vertex)
    - interface **IconVertex**
  - interface **GraphMLRepertoire** (also extends ElementCheckerDecorator, ElementFactoryDecorator)
  - interface **GraphMutation**
  - interface **GraphMutationListener**
  - interface **IconVertex**
  - interface **MutationFactory** (also extends ElementCheckerDecorator, ElementDialogDecorator, ElementFactoryDecorator)
    - interface **PeirceMutationFactory**
  - interface **PeirceElementChecker**
  - interface **PeirceElementDialog**

    - interface **PeirceMutationFactory**
- interface **BrowsingDecorator**
- interface ArchetypeVertex (also extends Cloneable)
    - interface **FamilyVertex** (also extends BasicInterface, Vertex)
        - interface **IconVertex**
    - interface **IconVertex**
    - interface Vertex
        - interface **FamilyVertex** (also extends BasicInterface)
            - interface **IconVertex**
        - interface **IconVertex**
- interface Element (also extends Cloneable)
    - interface ArchetypeVertex
        - interface **FamilyVertex** (also extends BasicInterface, Vertex)
            - interface **IconVertex**
        - interface **IconVertex**
        - interface Vertex
            - interface **FamilyVertex** (also extends BasicInterface)
                - interface **IconVertex**
            - interface **IconVertex**
    - interface **FamilyVertex** (also extends BasicInterface, Vertex)
        - interface **IconVertex**
    - interface **IconVertex**
    - interface Vertex
        - interface **FamilyVertex** (also extends BasicInterface)
            - interface **IconVertex**
        - interface **IconVertex**
- interface UserDataContainer
    - interface ArchetypeVertex
        - interface **FamilyVertex** (also extends BasicInterface, Vertex)
            - interface **IconVertex**
        - interface **IconVertex**
        - interface Vertex
            - interface **FamilyVertex** (also extends BasicInterface)
                - interface **IconVertex**
            - interface **IconVertex**
    - interface Element

        - interface ArchetypeVertex
            - interface **FamilyVertex** (also extends BasicInterface, Vertex)
                - interface **IconVertex**
            - interface **IconVertex**
            - interface Vertex
                - interface **FamilyVertex** (also extends BasicInterface)
                    - interface **IconVertex**
                - interface **IconVertex**
        - interface **FamilyVertex** (also extends BasicInterface, Vertex)
            - interface **IconVertex**
        - interface **IconVertex**
        - interface Vertex
            - interface **FamilyVertex** (also extends BasicInterface)
                - interface **IconVertex**
            - interface **IconVertex**
    - interface **FamilyVertex** (also extends BasicInterface, Vertex)
        - interface **IconVertex**
    - interface **IconVertex**
    - interface Vertex
        - interface **FamilyVertex** (also extends BasicInterface)
            - interface **IconVertex**
        - interface **IconVertex**
- interface Vertex
    - interface **FamilyVertex** (also extends BasicInterface)
        - interface **IconVertex**
    - interface **IconVertex**
- interface **EditingGraphMousePlugin.ShapeFunction**
- interface **ElementCheckerDecorator**
    - interface **GraphMLRepertoire** (also extends BasicInterface, ElementFactoryDecorator)
    - interface **MutationFactory** (also extends BasicInterface, ElementDialogDecorator, ElementFactoryDecorator)
        - interface **PeirceMutationFactory**
    - interface **PeirceMutationFactory**
- interface **ElementDialogDecorator**
    - interface **ElementFactory** (also extends Container)

  - interface **MutationFactory** (also extends BasicInterface, ElementCheckerDecorator, ElementFactoryDecorator)
    - ▹ interface **PeirceMutationFactory**
  - interface **PeirceMutationFactory**

- interface **ElementFactoryDecorator**
  - interface **GraphMLRepertoire** (also extends BasicInterface, ElementCheckerDecorator)
  - interface **MutationFactory** (also extends BasicInterface, ElementCheckerDecorator, ElementDialogDecorator)
    - ▹ interface **PeirceMutationFactory**
  - interface **PeirceMutationFactory**
- interface **GrammarModule**
- interface **GraphDecorator**
  - interface **IconGraphDecorator**
- interface **GraphMLElement**
- interface Icon
  - interface **VertexIcon** (also extends UberImageIcon)
- interface **IconConverter**
  - interface **IconModule**
- interface **IconConverterDecorator**
- interface **KeyDecorator**
- interface **MutationFactoryDecorator**
  - interface **ActionFactory** (also extends VisualizationControllerDecorator)
- interface **MutatorHandler**
- interface **PanelFactory**
- interface **PanelFactoryDecorator**
  - interface **ElementDialog** (also extends Container)
    - ▹ interface **PeirceElementDialog**
  - interface **PeirceElementDialog**
- interface **PeirceFactoryDecorator**
- interface Predicate

VilAug IconMessenger IconNet

  - interface **EdgePredicate**
  - interface **VertexPredicate**

- interface **RememberLast**
- interface **RepertoireDecorator**
- interface **UberImageIcon**
  - interface **VertexIcon** (also extends Icon)
- interface **VerboCentro**
- interface **VisualizationControllerDecorator**
  - interface **ActionFactory** (also extends MutationFactoryDecorator)
- interface **VisualizationViewerDecorator**

List of Figures

1.1 Physics Terminology Development . . . 4
1.2 Emergency Center using Visual Messenging . . . 7
1.3 Drive-In using Visual Messenging . . . 7

2.1 A gradation from Iconic to Sound-Based symbols . . . 20
2.2 Iconicity in form-meaning mappings . . . 20
2.3 SignWriting Example about Friendship . . . 23
2.4 The white flag and the situation of peace . . . 26
2.5 Conceptual Graph of “A cat is on a mat” . . . 29
2.6 Conceptual Graph with triadic relation . . . 29
2.7 MinSpeak: (Rainbow, Apple) can mean Red and (House, Apple) can mean Grocery . . . 32
2.8 Screenshot application VIL (by Leemans) . . . 34
2.9 Screenshot of Lingua . . . 37

3.1 Three Parallel Worlds . . . 45
3.2 Example of Menu Level Division . . . 46
3.3 Relation Inheritance . . . 49
3.4 Solution-based Graph Rewriting (from Chein & Mugnier, [37]) . 54
3.5 Ednoversion Example . . . 55
3.6 Unique Beginners in Leemans’ ontology . . . 59
3.7 Icon Inheritance Example . . . 60
3.8 Weak Inheritance Example . . . 61
3.9 Strong Inheritance Example . . . 62
3.10 Interface Inheritance Example . . . 63
3.11 Grammatical Categorization Example . . . 64
3.12 Visualization of Relational Concept Example . . . 66
3.13 DeleteConceptMutator . . . 67

4.1 Stakeholder Involvement Diagram . . . 73
4.2 System Overview . . . 74
4.3 From User Actions to System Functions . . . 75
4.4 Lingua & IconMessenger . . . 78
4.5 Visualizations WordNet . . . 79
4.6 Visualization Conceptual Graphs . . . 80
4.7 VilAug Package Overview . . . 82
4.8 Element (Factory) Package . . . 83
4.9 Model-View-Controller . . . 84

4.10 Registry Package . . . 85
4.11 Icon-based Extension of JUNG . . . 87
4.12 Icon-based VilAug Package . . . 88
4.13 GraphML Package . . . 91
4.14 Mutation Package . . . 92
4.15 Mutator Package . . . 93
4.16 DAG Package . . . 95
4.17 Predicate Package . . . 97
4.18 Grammar Module . . . 99
4.19 Peirce Package . . . 100

6.1 VilAug Introductionary Screenshot . . . 125
6.2 Typical icon messenger GUI (Lingua at the front) . . . 126
6.3 IconNet Browsing . . . 127
6.4 IconNet Icon Creation . . . 128
6.5 IconNet Relation Creation to Ontology . . . 129
6.6 Icon Menu after Relation to Ontology . . . 129
6.7 IconNet Relation Creation to Grammar . . . 130
6.8 Icon Menu after Relation to Grammar . . . 130
6.9 Relational to Normal Concept Creation . . . 131
6.10 Upgrading Relation . . . 132
6.11 Icon Menu after Relation Upgrade . . . 132

7.1 Stakeholder Involvement Diagram & Implemented Work . . . 147

8.1 IconNet & IconMessenger . . . 153

A.1 Cohn's CMSG Diagram . . . 155

List of Tables

1.1 Project Phases 5

2.1 DeFrancis' overview of Chinese Characters (only percentages are shown) 21
2.2 Levels of description suggested by Guarino 28
2.3 Categorization of Icons in VIL (columns are not related) 36
2.4 Categorization of Icons in Lingua (columns have no specific meaning) 38

7.1 Future Design Focus in regard to User Interface Design 144

B.1 Modifiers in Fillmore's Case Grammar, used in VIL 157
B.2 Primitive ACTs in Schank's Conceptual Dependency Theory . . 158
B.3 Categorization of Verbs in VIL 158
B.4 Categorization of Nouns in VIL 159
B.5 Categorization of Nouns in Lingua 159

List of Algorithms

1 GraphML File Format Syntax 89
2 Retrieve Icon Children from the IconManager 110
3 GraphMouse Plugin & Vertex Arguments 111
4 GraphMouse Plugin & Buttons 111
5 ChangeIconAction 112
6 ActivateIcon Mutator in Peirce Package 113
7 CreateVertex in VisualizationModel 113
8 VilAugIcon Properties 114
9 IconVertexIconAndShapeFunction 115
10 MutationLayout Methods 116
11 Calculation of Radius in Layout 116
12 PanelFactory Contract 117
13 Manifest File 117
14 Creation of Self-Executable Jar 118
15 Applet Evokation 118
16 Reading Graph Element Data 119
17 Applying Graph Element Data 119
18 IconModule Contract 120
19 GrammarModule Contract 121
20 File Reference Syntax 122

VilAug IconMessenger IconNet

Bibliography

[1] A. C. van Rossum, “Visual iconic language theory,” the theoretical part of van rossum’s master thesis., Delft University of Technology, Delft, The Netherlands, 2006.

[2] P. Leemans, *Visual Inter Lingua: VIL.* PhD thesis, Polytechnic Institute, Worcestor, USA.

[3] I. Roald, “Terminology in the making: Physics terminology in norwegian sign language,” *Vestlandet Resource Center for Deaf Education*, 2000.

[4] D. Poeppel and G. Hickok, “Introduction: Towards a new functional anatomy of language,” *Cognition, Elsevier*, vol. 92, pp. 1–12, 2004.

[5] K. Haberlandt, *Human Memory: Exploration and Application.* Allyn & Bacon, 1999.

[6] M. G. Dyer, “Distributed symbol formation and processing: in connectionist networks,” *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 2, pp. 215–239, 1990.

[7] A. R. Parker, “Evolving the narrow language faculty: Was recursion the pivotal step?.”.

[8] N. T. Cohn, “Eye (l) graefik semiosis!: A cognitive approach to graphic signs and ”writing”,” partial fulfillment of the requirements for the master of arts degree, May 2005.

[9] M. Gasser, “The origins of arbitrariness in language,” *Annual Conference of the Cognitive Science Society*, vol. 26, 2004.

[10] S. Wilcox, “Cognitive iconicity: Conceptual spaces, meaning, and gesture in signed languages,” *Cognitive Linguistics*, vol. 15-2, pp. 119–147, 2004.

[11] J. DeFrancis, “Visible speech: The diverse oneness of writing systems,” *University of Hawaii̇̈ Press*, 1989.

[12] R. C. Moore, “Removing left recursion from context-free grammars,” *Proceedings of the 1st Meeting of the North American Chapter of the Association for Computational Linguistics, ANLP-NAACL 2000. [Revision 30 January 2001]*, 2001.

[13] M. Johnson, “Memoization of top-down parsing,” *Computational Linguistics*, 1995.

[14] J. Heflin, "Owl web ontology language: Use cases and requirements," tech. rep., World Wide Web Consortium (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University), 2004.

[15] D. Chandler, "Biases of the ear and eye: "great divide" theories, phonocentrism, graphocentrism & logocentrism," *Web Document*, 1994.

[16] D. Chandler, "Semiotics for beginners," *Web Document*, 1994.

[17] S. Farrar and J. Bateman, "General ontology baseline," *OntoSpace Project Report*.

[18] O. Corcho, M. Fernández-López, and A. Gómez-Pérez, "Methodologies, tools and languages for building ontologies. where is their meeting point?," *Data & Knowledge Engineering*, vol. 46, pp. 41–64, 2003.

[19] J. F. Sowa, "Semantic networks." WWW Document, 12 2002.

[20] N. F. Noy and C. D. Hafner, "The state of the art in ontology design: A survey and comparative review," *American Association for Artificial Intelligence*, vol. Fall, pp. 53–71, 1997.

[21] H. Liu and P. Singh, "Omcsnet: A commonsense inference toolkit," *MIT Media Laboratory*.

[22] A. Gangemi, R. Navigli, and P. Velardi, "The ontowordnet project: Extension and axiomatization of conceptual relations in wordnet," *GWC 2004, Proceedings*, pp. 270–288, 2003.

[23] C. Bliss, "Blissymbolics communication international,"

[24] C. Beardon, "Discourse structures in iconic communication," 1975.

[25] O. Neurath, "International system of typographic picture education: Isotype," 1978.

[26] S. L. Tanimoto and C. E. Bernardelli, "The design and implementation of vedo-vedi, a visual language for human communication in the internet,"

[27] A. Stillman, "Kwikpoint,"

[28] A. Hollosi, "Alternative representations and beyond: A new proposal for a multi-sensory language interface (musli),"

[29] Insana, "Mediaglyphs,"

[30] A. Basu, "Sanyog: Visual communication interface,"

[31] B. Baker, "Iconic language design for people with significant speech and multiple impairments," *Prentke Romich Company*.

[32] S.-K. Chang, "Iconic language for mobile communication,"

[33] M. Pool, "The world language problem," *Utilika Foundation*.

[34] J. P. M. Gutiérrez, *Directed Motion in English and Spanish*, vol. 11 of *Estudios de Lingüística Española*. Universidad de Sevilla, España, 2001.

[35] P. H. Nguyen, "Relations as first-class citizens." NewsGroup Communication - Conceptual Graphs, may 2006.

[36] U. E. Priss, "Classification of meronymy by methods of relational concept analysis,"

[37] M. Chein and M. Mugnier, "Conceptual graphs are also graphs," tech. rep., LIRMM (CNRS and Université Montpellier II), 1995.

[38] N. Nomura and K. Muraki, "An empirical architecture for verb subcategorization frame: a lexicon for a real-world scale japanese-english interlingual mt," pp. 640–645.

[39] J. Anderson, "Concepts and consequences of case grammar," 2004.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, Addison-Wesley Professional, 1995.

[41] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall Series in Artificial Intelligence, Perason Education, Inc., 2th ed., 2003.

[42] A. C. van Rossum, "Designing the user interface," Master's thesis, TUDelft, 2006.

[43] B. Shneiderman and C. Plaissant, *Designing the User Interface*. Pearson Education, Inc., 4 ed., 2005.

[44] J. Morato, M. Ángel Marzal, J. Lloréns, and J. Moreiro, "Wordnet applications,"

[45] M. Spritzler, "Generifying your design patterns," *Web Document*, 2006.

VilAug
IconMessenger IconNet

Index

add concept, 65
add relation, 65
adjective inflection, 148
allowance parsing, 149
ancestor, 94
API, 41, 88
automatic grammar-classification, 69
automatic ontological knowledge acquisition, 140

canonical basis function, 48
child, 94
chiremes, 23
concept, 24
concept refining, 65
concept restriction, 65
conceptual graph, 29, 152
conceptual world, 44
conformity relation, 47
connectionist paradigm, 17
context free grammar, 22
CYC, 30

DAG, 68, 81, 95, 108
De Saussurean sign, 26
Deaf community, 23
denial parsing, 149
diagrammatic iconicity, 21
directed acyclic graph, 68
disjoint sum, 65
distinctions approach, 141
double articulation, 19, 23

Earley parser, 98
ednoversion, 55
external conformity relation, 47, 142

family analog, 94
file modularity, 90

generics, 150
gestural linguistics, 23
GNU General Public License, 77
GraphML API, 88
graphocentrism, 26

icon algebra, 32
iconic concept, 45
iconic relationship, 46
iconicity, 19
iconicon, 44
iconography, 44
iconology, 58
image caching, 87
induction, 142
inflection, 149
interface concept, 65
interface node, 63
internal conformity relation, 47
interpretant, 27
irredundant concept removal, 65
irredundant relation removal, 65

Java Runtime Environment, 150
javadoc, X
JRE, 150

lattice, 31
lazy image loading, 87
left-recursivity, 22
lightweight ontology, 29
Lingua, 34

meaningful units, 19
minimal functional units, 19
model-view-controller paradigm, 84
morphemes, 19

ontological model, 28, 41
ontology, 28
ordered grammar, 69
orthography, 19, 23

parent, 94

Peircean sign, 27
phonemes, 19
phonocentrism, 26
phonological loop, 16, 24

rebus principle, 19, 21, 32
reciprocal information, 68
redundant concept removal, 65
redundant relation removal, 65
referent, 27
regular language, 22
reification, 55, 141, 143
relation refining, 65
relation restriction, 65
representamen, 27

semiotics, 24
short-term memory, 16
software model, 41
stakeholder involvement diagram, 71
static type checking, 150
strict subsumption relation, 47
symbolic paradigm, 17

top-down parser, 22
translation by grouping, 149
type versus instance, 48

unique beginner, 53, 59, 76
universality, 33
unmodelled outerworld, 44
unordered grammar, 69
upgrade relation, 65
user grammar-classification, 69

verbal language, 18
verbocentrism, 25
VIL, 34
Visual Inter Lingua, 34
visual linguist, 1
visual linguistics, 5
visuolinguistic world, 44

WordNet, 30
working memory, 16