THE UNIVERSITY OF CALGARY

Constraint Graphs: A Concept Map Meta-Language

by

Robert Kremer

A DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

MAY, 1997

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a dissertation entitled "Constraint Graphs: A Concept Map Meta-Language" submitted by Robert Kremer in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

____

Supervisor, Brian Gaines
Department of Computer Science

____

Saul Greenberg
Department of Computer Science

____

Dickson Lukose
Department of Computer Science

____

Doug Norrie
Department of Mechanical Engineering

____

External Reader, Tom Carey
University of Guelph

____

Date

Abstract

Concept maps are visual graphs, consisting of nodes, which represent concepts, and arcs, which represent relationships between the concepts. Concept maps are used in a wide variety of disciplines because of their ability to make complex information structures explicit. There are a wide variety of concept mapping languages ranging from informal to formal; all share the same fundamental structure (nodes and arcs), but they vary in many ways, including degree of formality (typing), allowable component types, component graphical attributes, inclusion of contexts, and miscellaneous constraints. There is a large design space in concept mapping languages and each design is difficult and time consuming to implement. A simple way to prototype new concept mapping languages (and modify old ones) would be a very useful tool.

Constraint Graphs is a system designed to address this problem. The idea is to define a minimal concept mapping system, then let the end-user build a language based on these fundamental components. The base system is a very flexible concept mapping tool which is unconstrained except for the very fundamentals of graph theory (even n-ary arcs are allowed). In addition to "vanilla" nodes and arcs the system has an "isa" arc type which can be used to create a type lattice within the map. Constraints based on the type lattice allow a user the flexibility to use informal concept maps, to use a wide variety of formalisms, or to gradually constrain an informal concept map into a formal one.

The properties of such a "meta-visual language" avails itself to a variety of new applications including experimentation in concept map formalisms and powerful type checking tools for visual notations such as are used in knowledge modeling and decision making.

Acknowledgements

This work has benefited from the support of many people. Thanks to my supervisor, Brian Gaines, for his timely advice, encouragement, and criticism throughout the development of this work. Thanks are also due to the people at the Knowledge Science Institute lab and the Computer Science Department at the University of Calgary. These people have been an endless source of inspiration and good discussion throughout the work. They include Mildred Shaw, Roberto Flores-Mendez, Lee Chen, Saul Greenberg, Mark Roseman, Carl Gutwin, Ted O'Grady, Pim van Leeuwen, and Robin Cockett. Special thanks to Graham Birtwistle (now at Leeds University) who first encouraged me in the areas of object-oriented programming principles, formal methods, and type theory. I've also learned a lot recently from Dickson Lukose who has been on sabbatical at KSI from the University of New England, Australia.

I could not have completed this research without the support of Smart Technologies Inc. and the National Science and Engineering Research Council of Canada. Smart Technologies' president, David Martin, has been particularly supportive. In addition, the many technical discussions with Tom Fukushima of Smart Technologies have made an important contribution.

Thanks to my wife, Carolyn, for her encouragement and patience; and to my children, Cory, Chris and Sandra for their understanding of Dad's long hours at the computer. The encouragement and advice of my Dad and Mom has been greatly appreciated. My brother, Andy Kremer, made a tremendous contribution by proofreading my work.

This work is dedicated to my Dad, Mervin Kremer

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis is about visual knowledge representation languages. In particular, it focuses on a very flexible graphical language which can be constrained to model many other knowledge representation languages.

This chapter begins by briefly describing visual languages and a general kind of visual language: concept mapping languages. The many existing visual languages (and concept mapping languages in particular) are used in many different disciplines for widely diverse purposes. Computer support of concept mapping languages allows for easy editing and provides computational facilities such as those for the enforcement of syntactic and semantic constraints.

One of the problems with visual languages is the effort involved in creating a program to support each and every language. This thesis introduces Constraint Graphs, a computational environment to support the visual description of concept mapping languages that also uses the description to yield an implementation of the target language. Constraint Graphs also addresses the problem of lack of formal specification in visual languages. Several authors have addressed the specification issue, but have done so in a textual, predicate logic formalism. Visual languages should best be specified in visual languages. Constraint Graphs, by its own formal semantics and simple model, is a step toward a *visual* specification of concept mapping languages.

The last section of this chapter gives an overview of the software engineering aspects of the Constraint Graphs implementation.

1.1 A Note About Terminology

Concept mapping is related to several distinct research areas, including knowledge representation (Sowa 1991), graph theory (West 1996), graph grammars (Cuny, Ehrig & Engels 1994), and hypertext (Conklin 1987). Each of these areas deals with graphs – nodes with arcs that interconnect the nodes. Each of these areas is relatively independent, and all tend to use different terms for very similar, if not identical, concepts. The term *node* in concept mapping is called a *vertex* in graph theory, and a *concept* in many sub-disciplines of knowledge engineering. The term *arc* is called an *edge* in graph theory, a *relation* in many sub-disciplines of knowledge representation, and a *link* in hypertext theory.

It is often difficult or confusing, when describing these other research areas, to use non-native terms. This thesis therefore uses the terms *node, vertex,* and *concept* interchangeably, and does the same with *arc, edge, link,* and *relation*. The preferred terms, however, will be *node* and *arc*.

1.2 Visual Languages

The knowledge engineering community has developed a considerable number of formal knowledge representation languages including Ontolingua (Gruber 1995), CLASSIC (Borgida et al. 1989), Conceptual Graphs (Sowa 1984), and KIF (Genesereth & Fikes 1992). Some of these have graphical forms, such as KDraw (Gaines 1991b), which implements a completely visual version of CLASSIC. The graphical forms have considerable appeal because they present a relatively intuitive, two-dimensional view of the knowledge. In contrast, the purely textual languages tend to be useful only to experts who have invested considerable effort in understanding them.

```
define-concept[bird,(PRIMITIVE CLASSIC-THING Bird)].
define-concept[tubenose, (AND bird
                             (FILLS nostrils external-tubular)
                             (FILLS lives at-sea)
                             (FILLS bill hooked))].
define-concept[fulmar, tubenose, fulmar].
define-concept[albatross,(AND tubenose
                             (FILLS size large)
                             (FILLS wings long-narrow))].
define-concept[layson-albatross albatross].
define-concept[black-footed-albatross,
                         (AND albatross
                             (FILLS color dark))].
assert-rule[black-footed-albatross,
                         (FILLS name Black-footed-Albatross)].
```

Figure 1: Part of CLASSIC knowledge base on bird identification

For example, Figure 1 shows part of a CLASSIC knowledge base on bird identification that describes a section of the class-to-species hierarchy immediately above black-footed albatross. It describes several observable attributes that can be used to identify an observed bird successively as a tubenose (by having external, tubular nostrils; a hooked bill; and living at sea), as an albatross (by being large and having long, narrow wings), and finally as a black-footed albatross (by having dark coloration). To a non-specialist, Figure 1 seems rather opaque. On the other hand, Figure 2 (described in Section 2.3.1 on page 32), which is the visual version of the same portion of the same knowledge base, is more readily understood (Nosek & Roth 1990). Even a casual observer could guess that the graphical version is composed of a classification hierarchy on the left side and attributes of the of the classes on the right side. The detailed meaning of the classification may not be obvious to the casual observer; but to people knowledgeable in CLASSIC, the two dimensional layout makes all the relationships immediately and abundantly clear. Although the same information is conveyed by the textual version, the reader must exert considerably more cognitive effort to match up all the identifiers in order to mentally construct the hierarchy and relationships that are obvious in the graphical version.

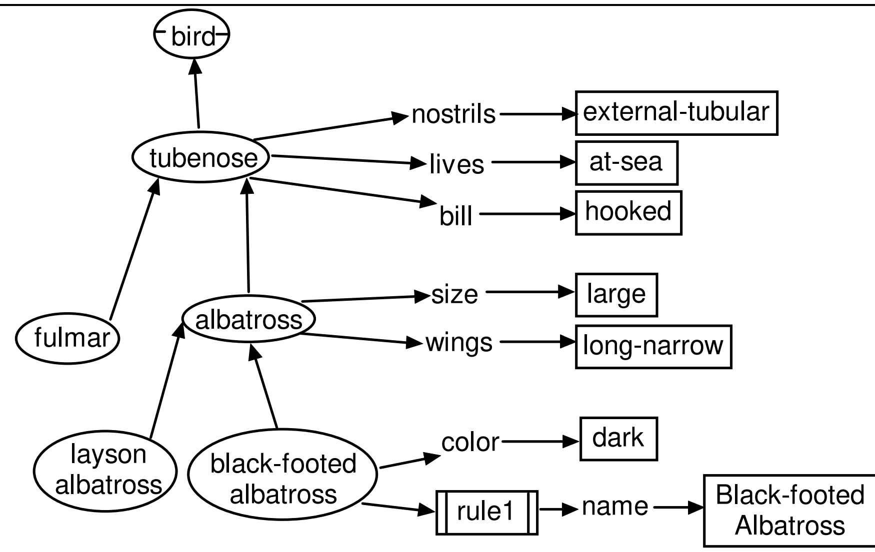


Figure 2: Part of a KDraw knowledge base on bird identification

There is still little hard evidence to support the value of graphical languages, or visual languages as they are now popularly referred to. But Nosek and Roth have undertaken empirical studies that indicate that (visual) semantic networks are more understandable than (textual) predicate logic (Nosek & Roth 1990). There are also large numbers of graphical, or visual, languages for a wide variety of purposes, including programming, program visualization, design of various kinds, decision making, network control, database query, and natural language generation and comprehension. Smith makes many arguments about the psychological motivations for using visual languages (Smith 1977) (see Section 2.1.1 on page 19).

Visual languages are useful for many purposes other than knowledge representation. Several authors (Myers 1990; Price, Baecker & Small 1993) have developed taxonomies for the ever-increasing variety of visual languages. These taxonomies range not only over the purpose of the visual languages, but also over their specification technique. This thesis focuses on knowledge representation – though it does diverge occasionally into other intended purposes. But this leaves still too wide a range of languages. So the focus is narrowed to a (still very broad) subset of visual languages, called *concept maps*.

1.3 Concept Maps

The term *concept map* is a general term describing a type of visual language. Concept maps consist of labeled nodes with (possibly labeled) arcs connecting the nodes. They are akin to *graphs* in mathematics.

There are a large number of concept mapping languages: of Myers' (1990) 14 taxonomy categories of visual languages, 6 categories are concept mapping languages. Furthermore, 24 of the 39 visual languages Myers studied were concept map languages. Concept maps have been used in many areas, including education (Lambiotte et al. 1984; Novak & Gowin 1984), management (Axelrod 1976; Hart 1977; Eden, Jones & Sims 1979; Banathy 1991), artificial intelligence (Quillian 1968), knowledge acquisition (McNeese et al. 1990), linguistics (Sowa 1984; Graesser & Clark 1985), programming (Burnett & Baker 1994), program design (Cox 1991; Booch 1994; Coad, North & Mayfield 1995), and program visualization (Myers 1990).

Concept maps are not just unconstrained graphs. Concept maps extend over a wide range of formality. Concept maps may be very informal and free form, such as "webs" used in education to allow students to describe their conceptual knowledge to teachers (see Figure 3, which shows a concept map developed by an unsupervised six-year-old boy using a concept mapping software package). Informal concept maps are very easy for people to create, because of the lack of constraints. They are therefore useful in education, in brainstorming, in the early stages of knowledge acquisition, and in any situation where the effort of conforming to a formalism may be too costly or time consuming, such as note-taking during a business meeting.

Concept maps may also be very formal and constrained. Figure 2 (page 4) is an example of a formal concept map. Formal concept maps are not as easy for humans to create and

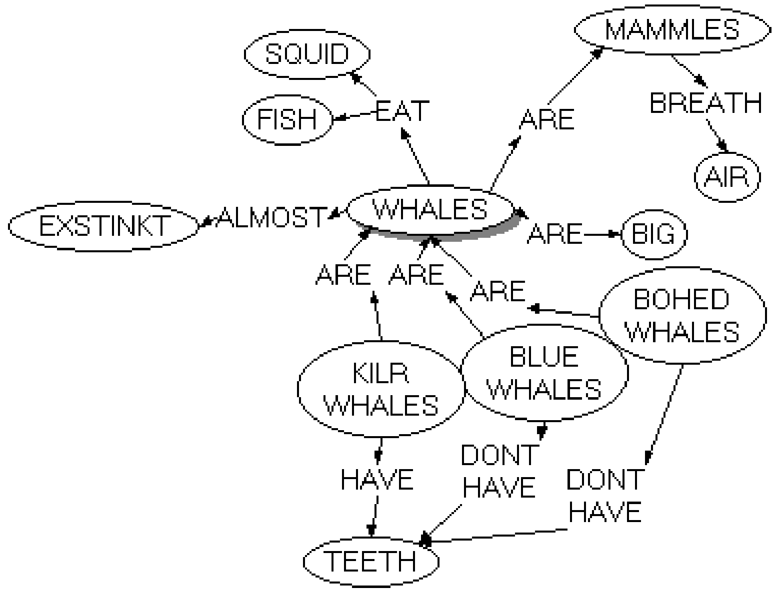


Figure 3: A concept map created by a six-year-old grade 1 student to describe his knowledge of whales.

usually require some degree of expertise, but their formality allows them to be interpreted by computers, which enables various forms of computational support. For example, formal concept maps are used to create expert systems (Gaines 1991b) and as complete programming languages (Smith 1977; Lukose 1993).

The formality of concept maps is not a discrete classification, but a continuum from informal to formal, with many shades of semi-formal languages in between. Semi-formal concept mapping languages strike a balance between human comprehension and the possibility of computational support. Examples of semi-formal concept mapping languages are gIBIS (Conklin & Begeman 1987), used in decision making, and object-oriented program design languages such as Booch notation (Booch 1994) and OMT notation (Coad, North & Mayfield 1995).

1.4 Supporting Concept Mapping Tools

Concept maps are difficult to draw using pen and paper because the two dimensional layout requires the author to predict the two dimensional extents in advance of starting to draw. Furthermore, additions and modifications to a graph often entail moving or rearranging large subsets of the map, which would mean much erasing and recopying if done with pen and paper. Computerized concept mapping tools help, not only by facilitating easy copy and move operations, but by enabling a broad range of constraints on layout, syntax, and semantics, depending on the language.

The previous section mentioned many different domains in which concept maps are used, and in most of these domains many different concept mapping languages are used. For example, software engineering alone uses object model notation, Booch notation, entity relation diagrams, data flow diagrams, Rumbaugh notation, structure charts, Petri nets, and others. It is clear that there are a vast number of concept mapping languages (and more to come in the future). If all of these are to be provided with computer support, it is clear that it can not be in the form of individual, specialized programs for each concept mapping language. The required software development effort is just too great. Many possible languages would not have computer support, and those that do would not be easy to modify in order to "evolve" the notation. Some way to ease the design of, development of, and experimentation with concept mapping languages must be found if concept mapping languages are to fulfill their potential as an expression medium.

Myers (1990), in his oft-cited (and three times published) taxonomy of visual languages, points out several problems with visual languages. Among those problems are:

- the difficulty in building editors and environments
- a lack of formal specification

These problems will be addressed in this thesis.

1.5 A Fast Framework for Concept Mapping Languages

It is not easy to create a visual concept mapping language tool. The complex graphics, direct manipulation user interaction model, object tracking, and constraint maintenance add up to a very complex system that is difficult and time consuming to specify, design, and build. The effort is not worth it for every potential visual language.

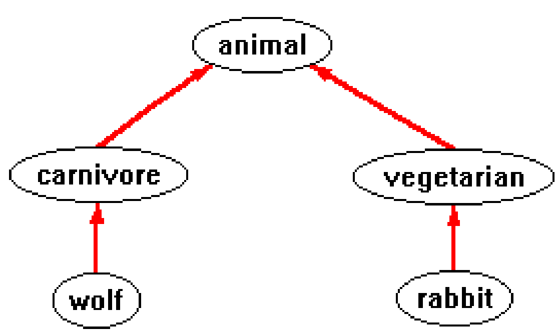
A framework for the fast development and modification of concept mapping languages would be very useful. Among the potential benefits are:

- *fast prototyping*: Concept mapping languages could be prototyped fast enough to enable empirical experimentation without the large commitment involved in specialized development efforts.
- *fast development*: Fully functional concept mapping languages could be developed much more quickly by basing them on the framework than they could be using a from-the-ground-up, custom develop effort.
- *intelligent support for users*: "Intelligent" add-ons (such as predictive commands, multi-user interfaces, and error correction) could be embedded in the framework. Such add-ons would be far too costly to develop for each and every individual custom development effort.
- *intelligent validation checks*: Constraint checking and map validation can be bundled into the framework. The otherwise prohibitively high cost of development of this feature would be amortized over all the languages implemented in the framework.
- *common interfaces*: All concept mapping languages implemented under a single framework would exhibit similar interfaces. Such a similarity is good for users since learning to use one concept mapping language would lead to positive learning transfer, and ease the learning of subsequent concept mapping languages written using the same framework.

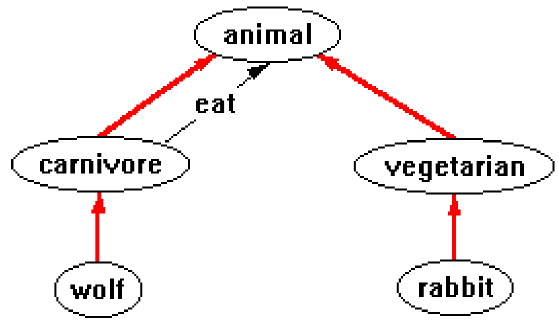
This thesis describes such a framework tool, called *Constraint Graphs*. Constraint Graphs provides an environment in which a large variety of concept mapping languages can be described using a simple concept mapping language. The new language description allows Constraint Graphs to behave like a program custom-made for that language, enforcing its syntactic (and some of its semantic) constraints.

Constraint Graphs is a distillation of many of the common features found in concept mapping languages. It is centered around a simple type theory. There are nodes and arcs which are mutually exclusive of one another. There may be any number of user-defined subtypes of nodes and arcs. The end points (or *terminals*) of arcs may be anchored on nodes or other arcs (arcs between other arcs are rarely, but sometimes, used in concept mapping languages). To describe an object type, the user merely draws it on the screen. To make one object a subtype of another, the user just draws a special type of arc (an *isa* arc) from the subtype to the supertype object. The types of the objects at each of an arc's terminals constrain all its subtype arcs to terminate on only subtypes of the corresponding objects.

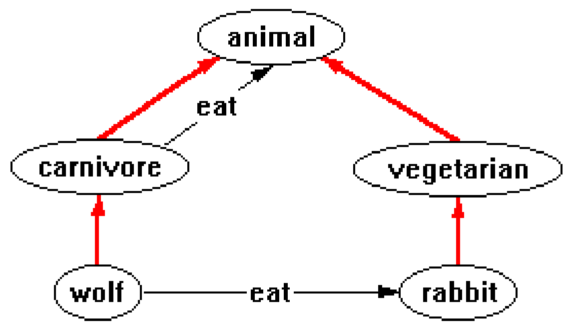
For example, one can define a domain of carnivores and vegetarians, where both are classified as subtypes of *animal*:



Here, the unlabeled arcs are *isa* arcs, representing the subtype relationship. In order to describe the fact that carnivores eat other animals, one creates an *eat* relationship between *carnivore* and *animal* by drawing in a new arc (labeled *eat*) between the *carnivore* node and the *animal* node:



Given this simple definition it is legal to draw an *eat* arc between *wolf* and *rabbit*:



because *wolf* is a subtype of *carnivore* and *rabbit* is a subtype of *animal*. If the user attempted to draw the arc in the reverse direction (from *rabbit* to *wolf*), the system would disallow it because *rabbit* is not a subtype of *carnivore*, and there is no other *eat* relationship that is legal.

In addition to just evaluating the legality of arcs, the system is capable of making type inferences: in the example above, the user need only indicate an arc from *wolf* to *rabbit*; the system can automatically type the relationship as *eat* because that is the only legal relationship between those two node types in the domain. Constraint Graphs can also associate attributes, such as surround shapes, colors, line types, and arbitrary user-defined attributes with objects, and can propagate these attributes and their values through the type (*isa*) hierarchy.

Using such simple techniques, Constraint Graphs can model many different concept mapping languages to provide a specialized interface for the target language. This ability to quickly prototype and implement visual languages shortcuts the expense of custom development efforts to allow the design and implementation of – and experimentation with – many new visual languages.

Chapter 6 describes many of the techniques used in Constraint Graphs for describing visual languages, and chapter 7 presents case studies of Constraint Graphs specification of several popular concept mapping languages (two knowledge representation languages and a semi-formal decision making language).

1.6 Concept Map Specification

Myers' wish for formal specification of visual languages is addressed by Constraint Graphs. Constraint Graphs itself is formally specified in the Z formal specification language (Hayes 1987; Spivey 1989). Chapter 5 describes Constraint Graphs' formal specification. Constraint Graphs' formal semantics, together with its very simple model, means that any language implemented on top of Constraint Graphs is well specified. A section in chapter 8 describes Constraint Graphs' ability to generate a Z specification of any language implemented in it.

While Constraint Graph's formal specification does not necessarily constitute a useful formal specification for Constraint Graphs-implemented languages, it does support the work of Crimi, Guercio, Nota, Pacini, Tortora, and Tuccit (1991) and Wang and Lee (1993) in visual language formal specification. Crimi et al describe a relational grammar formalism which serves to specify the syntax of a visual language. Wang and Lee describe a formal semantics for an *interpretation* that maps a visual language onto a domain language. Constraint Graphs' structure is consistent with both of these works, so avails itself to these formal descriptions as well.

Both the work of Crimi et al and Wang and Lee are textual predicate calculus descriptions of visual languages. However, it is intrinsically appealing to specify visual languages in a visual language. Constraint Graphs, by its very nature, does just that: concept mapping languages are described in terms of concept maps which have a formal definition.

1.7 A Principled Approach to System Design

Adopting a principled approach to system design and implementation is particularly important for a project like Constraint Graphs because it is more than just a program: it is a developing and evolving theory. For example, the first iterations of system design involved only node, binary arc, and *isa* objects as primitives; but as the design progressed it became obvious that the notions of n-ary (non-binary) arcs and contexts (graph partitions) needed to be included in the system, and that they could be included in a consistent manner. It seems inevitable that the need for other new facilities will arise as more and more concept map languages are implemented on top of Constraint Graphs.

The Constraint Graphs program is an object oriented program written in C++. It has been carefully designed to be very modular. It has a very distinct separation between large scale modules with a very defined interface between them: the Constraint Graphs "engine" (that implements the graph database and enforces constraints) is entirely separate from the user interface that actually draws the concept maps. The interface has been used *without* Constraint Graphs as a stand-alone, unconstrained, concept map drawing program. When Constraint Graphs and its interface are used together, the two components may be configured to run on different machines – due to the command-oriented interface between the two.

Constraint Graphs is designed for maximum flexibility. It is designed to run under a windowing operating system, but the details of the operating system have been abstracted away into a minimal windows library to enhance its portability. Furthermore, the interface is largely implemented within a "graphics" library, which allows a Constraint Graphs program to be configured as a stand-alone program; as a multi-user, real-time interaction program; or as a plug-in component to World Wide Web browsers, such as Netscape (again, either single- or multi-user).

At a finer level of granularity, the classes have been designed to faithfully conform to a type lattice, and the recently emergent *programming patterns* paradigm (Gamma et al. 1995; Pree 1995; Buschmann et al. 1996) has been adopted to increase the comprehensibility of the program by other programmers. Wherever possible, functionality (including those of the patterns) has been abstracted into independent classes and the "mix-in" (multiple inheritance) paradigm has been used to combine functionality.

Templates (type-parameterized classes) have been heavily used to abstract commonality between similar (but type-distinct) classes. For example the command objects of the command pattern (Gamma et al. 1995) have been implemented as a set of templates that can carry any object type in a type safe manner. The ANSII standard C++ STL (Standard Template Library) (Nelson 1995; Stepanov & Lee 1995) has also been extensively used for type safe container classes.

1.8 Objectives

The primary objective of this thesis is to provide the basis for a concrete theory of concept mapping language syntax by developing a software framework for the development and implementation of concept mapping languages. The work will narrow its focus somewhat to concentrate on knowledge representation languages.

The theory is a minimal, empirical theory. It is an *empirical* theory in that it attempts to describe a significant sub-class of visual languages known as concept mapping languages. It is *minimal* in the sense that it attempts to describe a very broad range of concept mapping languages without eliminating any languages unnecessarily – an attempt is made to avoid "over-specializing" the description while still capturing all of the essentials of concept mapping languages. It should be noted that the theory is meant to be criticized; it should be extended to encompass more (and new) concept mapping languages, and simplified to be as simple and as clear as possible. It is only *one* of many possible theories of concept mapping languages (Chapter 9 alludes to several other possible theories).

The framework acts as a concrete model of the theory, and should be flexible enough to encompass the fundamental syntax (and some semantics) of a wide selection of knowledge representation languages. The theory and framework will be tested by implementing language-specific environments for several existing knowledge representation languages. The framework-implemented language-specific environments should automatically enforce syntactic (and some semantic) constraints in a manner similar to the way a custom-coded environment would.

Since the concept mapping framework is intended to be a model of a developing theory, it is critically important that the framework be clearly described and cleanly constructed. This leads to two auxiliary objectives:

- The design should be formal and rigorous where appropriate
- The framework should be developed in a principled manner

To provide a formal, rigorous, and unambiguous description, parts of the framework will be formally specified in the Z specification language. Note that not all of the framework should follow such rigor: the actual concept map drawing tools are not (and should not be) a part of a theory of concept maps, so the effort of rigorous specification is unwarranted.

It does no good to have a rigorous specification if the program developed from the specification is not a reliable implementation of the specification. It is therefore considered important to develop the framework in a principled manner, taking advantage of current technology such as design patterns (Gamma et al. 1995; Pree 1995) and the C++ Standard Template Library (STL) (Nelson 1995; Stepanov & Lee 1995).

Furthermore, a straightforward correspondence between specification and implementation is also considered important, which entails favoring implementation decisions which enhance the direct correspondence to the specification over efficiency and similar considerations.

Further details about these objectives may be found in Section 4.1 on page 56.

1.9 Summary

This chapter has briefly introduced visual languages and concept maps, especially with respect to visual knowledge representation languages. Concept mapping languages comprise a broad subset of visual languages, and are useful in many different disciplines for many different purposes. Concept maps range from the informal (free-form for ease-of-use) to the very formal (very constrained for computer support). Computer support for concept mapping is a real need to allow for easy editing and for explicit enforcement of syntactic and semantic constraints.

However, one of the problems with visual languages is the difficulty of writing programs to support them. Constraint Graphs addresses this problem by providing a visual (concept mapping) environment in which new concept mapping languages can be described. Constraint Graphs can then use such a description to "implement" the new language.

Another problem with visual languages is the lack of formal specification. This problem has been addressed by several authors but by using a textual predicate logic. Constraint graphs addresses the specification issue by providing a simple "platform" theory that is visually expressed in terms of concept maps. Visual languages are probably best specified in visual languages. Constraint Graphs is a start in this direction.

Since the Constraint Graphs program is seen as supporting an evolving theory, it is critical that it be designed to be as flexible and easy-to-understand as possible. The object-oriented software development paradigm, formal specification, and the recently emergent programming pattern paradigm have been adopted to provide this flexibility in design and implementation.

The primary objective of this thesis is to develop a software framework for the development and implementation of a variety of concept mapping languages. The framework should be able to serve as a test bed for a developing theory of concept mapping languages. To accomplish this goal, the framework must be rigorously specified where appropriate and implemented in a principled manner.

1.10 Thesis Structure

Chapter 2 describes visual languages and concept maps in detail and examines concept mapping's relationship to several other areas of study such as relational grammars and graph grammars.

Chapter 3 deals with types and the object oriented programming paradigm, then shows how these are related to concept maps. Concept maps as hypergraphs are discussed and current work on flat-typed hypergraphs is slightly extended to explain hypergraphs with types arranged in a type lattice. There is a short section at the end of Chapter 3 on formal specification by way of an introduction for the specification later in the thesis.

Chapter 4 describes the requirements analysis for the Constraint Graphs project. Most of this work deals with the abstract implementation of the Constraint Graphs database, which is responsible for the graph operations and constraint maintenance. The interface requirements are also described, but only briefly.

Chapter 5 deals with the formal Z specification of a Constraint Graphs system. Only the abstract database and graph operators are specified: the interface specification is much less formal since it is seen as just a replaceable component.

Chapter 6 describes the actual implementation based on the specification. It describes the separation of the main modules, the interface and actual Constraint Graph class libraries, the intended usage scenarios, and several important technical details.

Chapter 7 describes three concept mapping languages, and how they are implemented on top of Constraint Graphs. The implementation of gIBIS, a very simple decision making language, is described in detail. This serves to guide the reader through a very simple implementation to gain a flavor of Constraint Graphs definitions. Subsequently, the more complex and harder-to-implement knowledge representation languages, Conceptual Graphs and KDraw are described, building on the previous language descriptions.

Chapter 8 describes Constraint Graph's Z translator (which can translate a concept map definition into a Z specification); the World Wide Web browser-embedded version of Constraint Graphs; and a multi-user version of Constraint Graphs.

Chapter 9 is an evaluative conclusion and includes directions for future work.

Chapter 2

Background: Visual Languages and their Domains

This chapter focuses on visual languages; in particular, the field is narrowed (slightly) by concentrating on *concept map*-based visual languages. Here, a concept map consists of nodes and arcs and possibly other constructs, such as contexts. Thus, visual languages based on matrices (spreadsheets), forms (query by example), purely spatial relationships (iconic sentences), and simple text layout (outlining tools) are eliminated. All of the example languages used in this chapter either fall into the category of concept maps, or have concept map languages which implement them. For example, Conceptual Graphs has both a graphical form (concept map) and a "linear form" (textual); and Gaines (1991b) has developed a visual language for CLASSIC, called KDraw as part of his Knowledge Representations System (KRS).

Although the focus of this thesis is on formal and semi-formal visual knowledge representation, informal representations are not completely neglected, for they are often useful (Kremer 1994; 1995):

- in preliminary knowledge gathering where the appropriate formal structure is not apparent,
- when preliminary work is carried out by domain experts who are novices at knowledge representation techniques, or
- when the situation demands a relaxation of formal discipline (such as during an intense meeting)

In addition, Novak and Gowin (1984) describe the use of informal concept maps in education.

2.1 Visual Languages

A visual language, as the term is used here, is any form of communication that relies on two- or three-dimensional graphics rather than simply (relatively) linear text. Even pure text is often supplemented with visual properties such as indentation and even font styles such as bolding and italics. For example, textual programming languages are far more readable if indented to indicate the hierarchical structure; the following code:

```
void makeupper(char* buf) {char* i; for (i=buf; *i; i++) {*i
= toupper(*i);}}
```

is much more difficult to comprehend than the same code with appropriate indentation:

```
void makeupper(char* buf)
      {
      char* i;
      for (i=buf; *i; i++)
            {
            *i = toupper(*i);
            }
      }
```

In spite of the visual aids, this is still considered a textual language. Visual languages all involve pictures of some sort – whether as simple as nodes and connecting arcs, or as complex as nested, multi-dimensional sentences of iconic pictures. Of course, visual languages often involve text, but text's role is usually limited to labels or brief explanations or annotations.

2.1.1 The Value of Visual Languages

In certain circumstances, visual representations have significant advantages over (relatively) linear textual representations. Besides the old adage "a picture is worth a thousand words", Nosek and Roth (1990) have concluded that semantic networks (graphs) are more "computationally efficient" (understandable) than predicate logic (linear text). Two dimensional representations of programs, such as flow charts and even the indentation of textual programming languages, are extremely helpful to aid program understanding (Smith 1977). There are many visual programming languages and languages for program visualization (Myers 1990; Marks 1991; Price, Baecker & Small 1993) as well as large numbers of visual languages for other purposes such as decision making (Conklin & Begeman 1987), knowledge representation (Sowa 1984; Gaines 1991a), geographic information system queries (Calcinelli & Mainguenaud 1994; Aufaure-Portier 1995), and database queries (Constabile & Catarci 1995), to name just a few.

It can be argued that not all people are good at reading graphical representations of data and knowledge. The point seems obvious, but Petre and Green have found evidence that "seeing" the information in a graphical display is an acquired skill (Petre & Green 1993). In contrast to Nosek and Roth's study, Petre and Green's study found that subjects could solve problems faster with their textual languages than with their graphical languages; however, they conclude that although people seem to have to work harder to read graphical information, the graphical notations may prove much richer for expert users. Indeed, there is still not sufficient evidence to evaluate the usefulness of visual languages. Myers (1990) notes the lack of evidence:

> There are not many Visual Languages that would be generally agreed are 'successful', and there is little in the way of formal experiments or informal experience that shows that Visual Languages are good. It would be interesting to see experimental results that demonstrated that visual programming techniques or iconic languages were better than good textual methods for performing the same tasks. Metrics might include learning time, execution speed, retention, etc. Fortunately, preliminary results are appearing for the advantages of using graphics for teaching students how to program. (p. 117)

Regardless of the lack of firm evidence, there is much intuitive appeal to graphical languages. Smith (1977) describes the psychological motivations for using visual languages for programs and data. Among Smith's many arguments are:

- There is evidence to support the notion that abstract reasoning is pictorial in nature.
- The two- and three-dimensional nature of visual reasoning may be more efficient than linear verbal language.
- Visual organizations are an extremely efficient "chunking" method to increase the limit capacity of short-term memory.
- Thought operations may be transformations of images.
- Images are usual analogous to tasks, while linear languages are mostly *Fregean* (bear no analogous relation to the task they describe). Analogical systems are more accurate than Fregean systems.
- Some experiments indicate that the mind is capable of storing sensory data in great detail.

One may also glean some empirical evidence for the power and convenience of visual languages from the experience of Charles Sanders Peirce. Sowa (1984) based his Conceptual Graphs work on Peirce’s much earlier work:

> "The standard notation for symbolic logic was developed by mathematicians like Peano, Russell, and Whitehead, who patterned it after algebra. But one of the pioneers of logic, Charles Sanders Peirce, switched to a graph notation after making major contributions to the linear form. His *existential graphs*, which he called ‘the logic of the future,’ form the logical basis for conceptual graphs." (p. 138)

While Peirce's graphical notations (visual logic languages) have not been well-accepted by mainstream logicians, his invention and use of them argue for their utility for this "pioneer of logic". Peirce himself writes:

> "I do not think I ever *reflect* in words: I employ visual diagrams, firstly because this way of thinking is my natural language of self-communion, and secondly, because I am convinced that it is the best system for the purpose [Ms 620, p. 8]" (Roberts 1973, p. 126)

Roberts (1973), in his book about Pierce's existential graphs, argues for the usefulness of the visual logic language:

> "Logic diagrams these days are certainly less familiar than algebraic notations of logic. But are they really too cumbersome? Would it be fair to say that they are "less perspicuous than the usual" notations, as some say of the Polish notation (Church 1956)? Well, if a facile and perspicuous notation is one that can be quickly learned and easily manipulated, then years of experience with university students have convinced me that EG is the most perspicuous, and *Principia* notation the least. The unusual ease with which inferences can be drawn in EG is something of an unexpected bonus." (p. 126)

The current popularity of direct manipulation interfaces (Shneiderman 1986) is completely compatible with visual languages since many are subject to manipulation with a mouse or other pointing device.

2.1.2 The Meaning of Visual Languages

Like all languages, visual languages can only be reliably interpreted if they have a formal syntax and semantics, though many visual languages lack semantics or both (Myers 1990). Visual languages seem to lag well behind textual languages in this regard. Most formal textual languages (programming languages) are well understood because of the amount of work put into investigating their formal semantics. Little equivalent work has been done for visual languages. This work is described in this section.

Crimi et al (1991) describe the application of relational grammars to multi-dimensional (visual) languages in order to define a visual syntax. In their formulation, a context-free relational grammar is a tuple,

$$G = (V_N, V_T, V_R, S, P, R)$$

where:

- V_T is a finite set of terminal symbols.
- V_N is a finite set of non-terminal symbols.
- V_R is a finite set of relation symbols, $V_R = \{r_i | 1 \leq i \leq k\}$; for each r_i its arity $a(r_i)$ is defined.
- S is the starting symbol.
- P is a finite set of productions, where each production has the form A::=M,R where: $A \in V_N$; M is a multiset of symbols in $V_N \cup V_T$; and R is set of *constraints* over the symbols in M which must hold for a production to be applied.
- R is a set of *evaluation rules* used to validate constraints.

The constraints, R, in the productions, P, are defined by the evaluation rules, R. For example, the set of all possible untyped directed graphs is defined by (V_N, V_T, V_R, G, P, R), where ("O" represents a node and "→" represents a directed arc):

```
VN = {G}
VT = {O,→}
VR = {start,end}
P =   1. G::={O}
      2. G::={O,G}
      3. G::={→,G} {start(→,G), end(→,G)}
R =   1. start(→,G) :- (G ⇒ {O}), start(→,O)
      2. start(→,G1) :- (G1 ⇒ {O,G2}), start(→,O)
      3. start(→,G1) :- (G1 ⇒ {O,G2}), start(→,G2)
      4. start(→1,G1) :- (G1 ⇒ {→2,G2}), start(→1,G2)
      5. end(→,G) :- (G ⇒ {O}), end(→,O)
      6. end(→,G1) :- (G1 ⇒ {O,G2}), end(→,O)
      7. end(→,G1) :- (G1 ⇒ {O,G2}), end(→,G2)
      8. end(→1,G1) :- (G1 ⇒ {→2,G2}), end(→1,G2)
```

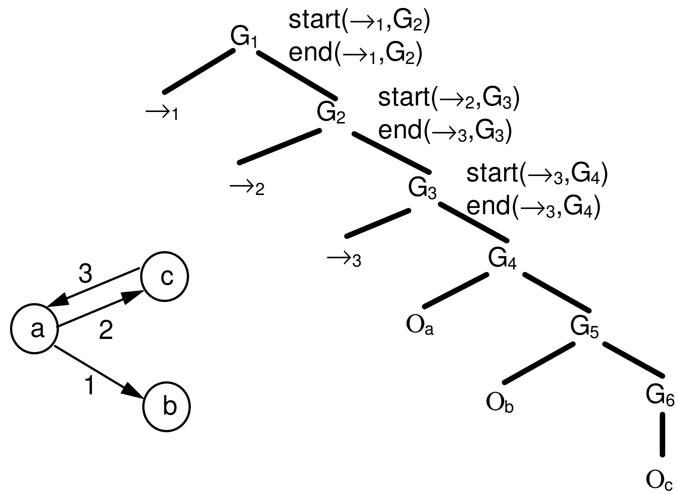


Figure 4: A directed graph and its parse tree (Crimi et al. 1991)

Production 1 says that a single node is a legal graph. Production 2 says a single node can always be added to a graph. Production 3 says a directed arc can only be added to a graph if it starts and ends in the graph.

The evaluation rules, R, specify how the start and end relations are to be interpreted. They are defined recursively in a Prolog-like language. For example, Rule 1 states that a directed arc starts on G if G matches a single node (G $\Rightarrow$ {O}), and the arc starts on that node (start($\rightarrow$,O)). Rules 2 and 3 state that a directed arc starts on G_1 if G_1 matches {O,G_2} and the arc starts on the node (start($\rightarrow$,O)) or on a node in G_2 (start($\rightarrow$,G_2)). Figure 4 shows a simple directed graph and its parse tree according to this relational grammar.

In contrast, Wang and Lee (1993) do not concern themselves with syntax, but go a step in the direction of semantics. They describe a formal semantics for visual reasoning. In their formulation, graphs are given meaning by an *interpretation* that maps the graphical domain to an application domain:

$$\boxed{\text{graphical domain}} \xrightarrow{\text{interpretation}} \boxed{\text{application domain}}$$

A *graphical signature*, Σ, is a quadruple (T, $\leq$, F, P) where:

- (T, $\leq$) is a partially ordered set of sorts
- F is a set of function symbols of the form $s_1 \times s_2 \times \ldots \times s_n \rightarrow s$, where $s_i, s \in T$
- P is a set of predicate symbols of the form $s_1 \times s_2 \times \ldots \times s_n$, where $s_i \in T$

Thus, T is the set of graphical object types in the graphical domain (e.g.: circle $\leq$ closed_curve) , and F and P are just the set of functions (e.g.: overlap: circle $\times$ circle $\rightarrow$ closed_curve) and predicates (e.g.: overlapping: circle $\times$ circle) defined on them. A *domain language* has a similar signature, Σ_D, and an interpretation, I: $\Sigma \rightarrow \Sigma_D$, defines functions from the element of T, F, and P in Σ to the elements of T_D, F_D, and P_D in Σ_D respectively (where each of the functions must preserve the partial order, type, and domain, respectively). Table 1 gives an example interpretation from a graphical domain of circles and points to an application domain of sets and elements.

2.1.3 Classification of Visual Languages

There are several taxonomies of visual languages, though most focus on specific domains. For example, Price, Baecker, and Small (1993) focus only on programming visualization languages. Myers (1990) concentrates on the area of *visual programming* and *program visualization* and describes three separate taxonomies. Visual programming is the specification of programs using graphical techniques while programming visualization is using graphical techniques to explicate the structure of a (not necessarily visual) program. Myers' first taxonomy is about programming systems and is based on the following three dimensions:

- visual programming vs. not visual programming: differentiates true visual languages from traditional text-based languages.

- example-based programming vs. not example-based programming: differentiates systems that attempt to infer programs from one or more examples from those that require explicit specification.
- interpretive vs. compiled: differentiates systems that can be directly executed from those that require an intermediate translation before they can be executed.

Myers' second taxonomy is of particular interest to this work and classifies systems into 14 categories of *specification technique*. Table 2 lists Myers' specification techniques showing the six types that are kinds of concept maps. It is interesting to note that not only are six of Myers' 14 classifications different forms of concept maps, but that fully 25 of his 39 non-textual-language examples fall into these six concept map categories.

Table 1: An interpretation from the graphical domain to the domain of set theory

Σ	T	Point, Circle
	$\leq$	$s \leq s$ for all $s \in T$
	F	a,b:Point, A,B:Circle
	P	in_1:Point $\times$ Circle; in_2:Circle $\times$ Circle
Σ_D	T_D	Element, Set
	$\leq_D$	…
	F_D	a,b:Element, A,B:Set
	P_D	$\in$:Element $\times$ Set; $\subseteq$:Set $\times$ Set
I	I_T	Point $\rightarrow$ Element, Circle $\rightarrow$ Set
	I_F	$A \rightarrow A$, $B \rightarrow B$, $a \rightarrow a$, $b \rightarrow b$
	I_P	$in_1 \rightarrow \in$, $in_2 \rightarrow \subseteq$

Myers' third taxonomy divides program visualization systems according to whether they represent code, data, or algorithms.

There are other taxonomies of visual languages, but Myers' are the most well-known, and the others tend to specialize within a specific area. For example, Price, Baecker and Small quantitatively grade their 12 example program visualization languages in each of over 30 leaf-node categories arranged in a four level hierarchy (Price, Baecker & Small 1993).

Table 2: Myers' taxonomy of specification techniques (Myers 1990)

	Textual languages
Concept maps	Flowcharts
	Flowchart derivatives
	Petri nets
	Data flow graphs
	Directed graphs
	Graph derivatives
	Matrices
	Jigsaw puzzle pieces
	Forms
	Iconic sentences
	Spreadsheets
	Demonstrational
	None

2.2 Concept Maps

Concept map is a very general term that describes a simple and intuitive type of visual language. For human users, concept maps tend to make the structure of a body of knowledge much more salient than other forms of knowledge representation such as pure text and predicate logic (Nosek & Roth 1990). Concept maps have been used in education (Lambiotte et al. 1984; Novak & Gowin 1984), in management (Axelrod 1976; Hart 1977; Eden, Jones & Sims 1979; Banathy 1991), in artificial intelligence (Quillian 1968), in knowledge acquisition (McNeese et al. 1990), in linguistics (Sowa 1984; Graesser & Clark 1985) and for many other varied purposes. Concept maps can (and have) been used to represent knowledge at the very informal level, such as for "brainstorming", as well as at the very formal level, such as executable expert systems (Gaines 1991b) or graphic forms (Ellis & Levinson 1992; Eklund, Leane & Nowak 1993) of Conceptual Graphs (Sowa 1984) (see also (Kremer 1994; 1995)).

Concept maps are graphs (in the mathematical sense of the word) consisting of nodes with connecting arcs, which represent relationships between nodes (Lambiotte et al. 1984). The nodes are labeled with descriptive text, representing the “concept”, and the arcs are often labeled with a relationship type. Nodes may be represented using distinct visual attributes, such as shape and color, to distinguish node types; arcs may be similarly distinguished. Thus, concept maps model a simple graph theory. The notion of concept map may be extended to include *contexts*, boxes that contain a subgraph (sub-concept map) of the graph (concept map), which models *partitions* in graph theory.

Figure 5 is an example of an informal concept map developed by a group during an interactive brainstorming session about a new network in a large company. In contrast, Figure 6 (page 35) shows a formal concept map, called a Conceptual Graph (Sowa 1984), which models the sentence "Tom believes Mary wants to marry a sailor" in a way that is amenable to straightforward, unambiguous interpretation by a computer program.

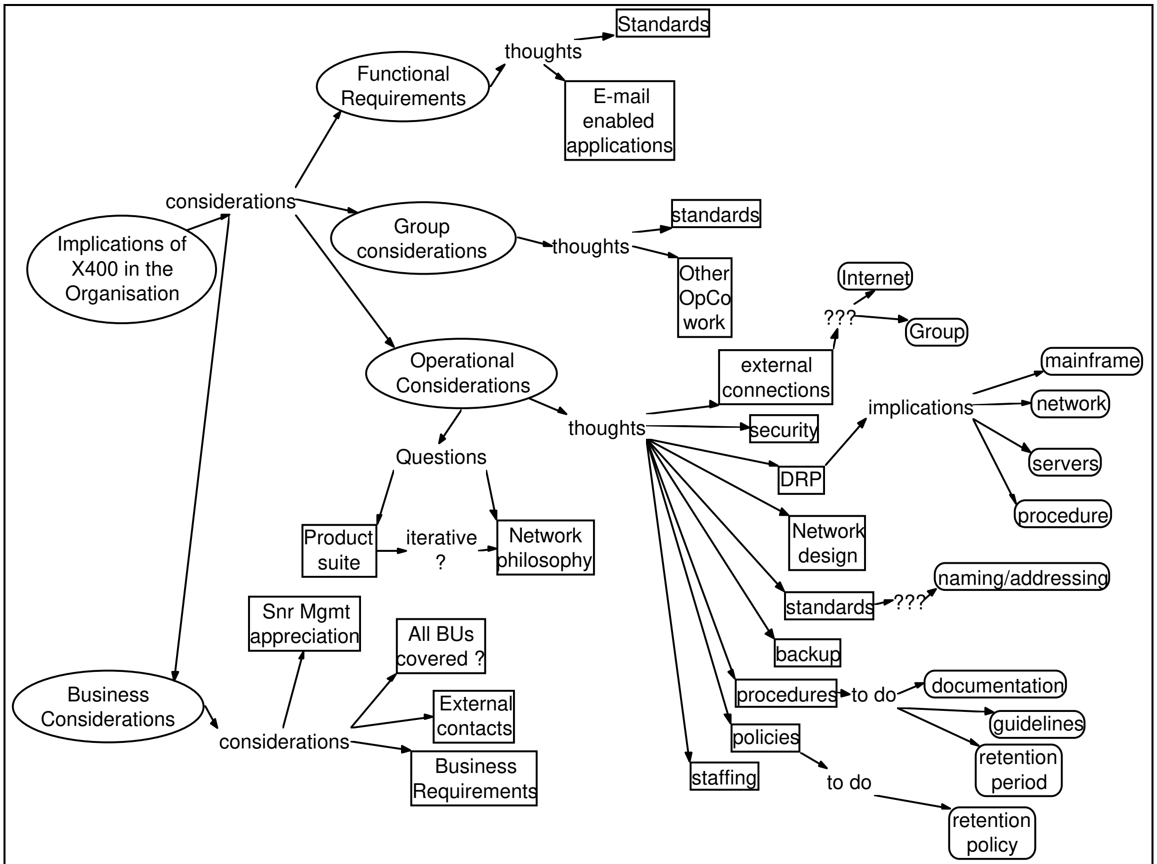


Figure 5: A concept map developed by a three member group brainstorming network design (Kremer 1993).

Table 2 (page 27) shows where concept maps fit into Myers' taxonomy of visual languages, subsuming six of his 14 categories and constituting 24 of his 39 example visual languages. Any of the various forms of concept maps may be described in terms of Crimi et al's relational grammars (Crimi et al. 1991) in a manner very similar to the example given in Section 2.1.2. This example, however, only considers directed graphs, while concept maps may contain directed, undirected, and bi-directional arcs – or a mix of these. The example is also overly simple in that it assumes only untyped nodes and arcs, while concept mapping languages usually include typed nodes and/or typed arcs where the rules include constraints on the interconnections based on type.

In terms of Wang and Lee's semantics (Wang & Lee 1993), the term *concept map* is too general to be assigned a mapping to a specific application domain: there are many concept mapping languages which each map to their own set of application domains. However, one can describe the graphical signature, Σ, of concept maps at a high level (*top* and *null* are the universal type and the absurd type, respectively):

Σ: $\mathcal{T}$ top, pos_int, graph, node, arc, null

$\leq$ graph $\leq$ top, node $\leq$ graph, arc $\leq$ graph, null $\leq$ node, null $\leq$ arc, pos_int $\leq$ top, null $\leq$ pos_int

$\mathcal{F}$ anchored_to: arc $\times$ pos_int $\rightarrow$ node

$\mathcal{P}$ anchored: arc $\times$ pos_int $\times$ node

This description specifies that graphs contain nodes and arcs, but no common subtype between the two. It specifies that arcs may be connected only to nodes, but this may not necessarily be a restriction in all concept mapping languages (arcs may sometimes connect to other arcs). Furthermore, Σ is quite general in that it allows for arcs of arbitrary arity, where most (but not all) concept mapping languages restrict themselves to binary arcs. The function *anchored_to* could be used to determine the node at the *i*th terminal of an arc; the predicate *anchored* can be used to test whether the *i*th terminal of an arc is attached to a certain node. If this description were to be extended to include contexts, a (non-universal) common supertype of *node* and *arc* would have to be introduced to allow both to be included in partitions.

Lambiottte, Dansereau, Cross, and Reynolds (1984) classify "multirelational semantic maps" into four categories:

- Spatial layout and location systems: These systems rely on spatial proximity, spatial relationship (e.g.: above or below), or boundary lines to convey relationships among items. Layout systems are limited in complex domains because it is difficult to convey more than a very few relationship types spatially.

- Node based systems: These systems generally use node attributes (e.g.: shape, color, or style) to distinguish a variety of nodes types to convey information. If arcs are used at all, the relationship they represent is ambiguous, or must be inferred from the adjoining node types.
- Link-based systems: These systems convey almost all information in arcs which may be labeled and/or otherwise distinguished by attributes such as style, thickness or color to convey their type either canonically (drawn from a set of predefined types) or idiosyncratically (ad-hoc types). If nodes are present at all, they contain only the data, and are uniform in appearance.
- Hybrid systems: These systems take advantage of spatial layout, node type, and link type to convey a much richer set of semantic information.

The first category, spatial layout, may not be considered a concept map at all. The other three categories constitute concept maps with typed nodes but untyped arcs, concept maps with untyped nodes but typed arcs, and concept maps with both typed nodes and typed arcs. Although spatial layout does not usually effect the explicit meaning in concept mapping languages, it does have an effect on the understandability of a particular map. Concept mapping languages with context do use spatial layout in that containment in contexts usually conveys semantic meaning.

The remainder of this chapter focuses on examples of concept mapping languages. These include two languages for knowledge representation and one language for decision making.

2.3 Visual Knowledge Representation Languages

Almost any language (including English) can be conceived of as a knowledge representation language. But the emphasis here is on formal and semi-formal concept mapping languages with (almost) unambiguous semantics. That is, this thesis does not focus on natural languages (like English), but instead concentrates on "artificial" languages like CLASSIC (Borgida et al. 1989) and Conceptual Graphs (Sowa 1984) whose primary intent is to render knowledge in a form amenable to computational support. CLASSIC and its visual counterpart, KDraw (Gaines 1991b), seek to capture knowledge such that inferences can be made to classify the domain elements. One of Conceptual Graphs' goals is to represent natural language expressions in a graphical form, but sufficiently disambiguated so as to render the representations amenable to computer processing. These are "pure" knowledge representation languages. Also of interest are languages with less lofty goals – some of these are discussed in subsequent sections.

In order to illustrate visual knowledge representation languages, this section describes two very different visual knowledge representation languages: Gaines' KDraw, a visual form of CLASSIC; and Sowa's Conceptual Graphs, which has both a graphical and linear (textual) form. Coincidentally, these two languages are both used as examples in Chapter 7.

2.3.1 KDraw

CLASSIC (Borgida et al. 1989) is a textual knowledge representation language in the KL-ONE family of languages. CLASSIC is an "open world" database, as opposed to the relational data base model which embraces a "closed world" assumption (if it is not in the database it doesn't exist). Users can populate the CLASSIC database with *concepts*, *primitives,* and *individuals* and assign *roles* (like fields in relational databases, or slots in object oriented systems) to the concepts, primitives, and individuals. The roles can be constrained as to type (concept conformance) and arity, and can be assigned values in the form of individuals. For example, a user can add the concept CRIME to a CLASSIC database as follows:

```
define-role[perpetrator]
define-role[victim]
…
define-concept[CRIME,
      (PRIMITIVE
            (AND  (AT-LEAST 1 perpetrator)
                  (ALL perpetrator PERSON)
                  (AT-LEAST 1 victim)
                  (AT-LEAST 1 site)
                  (AT-MOST 1 site))
      crime)].
```

CLASSIC works by classifying *individuals* as members of one or more *concepts* in a concept hierarchy. Information about concepts and individuals is encoded as *roles* (attributes or slots) and associated *values*. CLASSIC matches the roles and values of individuals to infer which concepts an individual belongs to. Once an individual has been classified, *rules* associated with the concepts may be fired to further instantiate knowledge about the individual (by filling in roles and role values).

There has been extensive work on complexity of the subsumption algorithms in KL-ONE languages (Donini et al. 1995). The subsumption algorithm is important because subsumption calculation is the basis of all computation in KL-ONE languages. It turns out that extension of a base level version of KL-ONE (composed of conjunction, limited negation, universal quantification, limited role existential quantification, and numeric role filler limits) leads to worst-case complexities of NP-hard or worse. Fortunately, subsumption seems to be computationally tractable in practical situations.

Gaines has developed a visual version of the CLASSIC language, called KDraw in which the user can specify and run a complete expert system (Gaines 1991b). A small excerpt from a knowledge base about birds is illustrated in Figure 2 on page 4. The ellipse with tick marks identifies *bird* as a primitive concept. The unmarked ellipses labeled *tubnose*, *albatross*, *fulmar*, *layson albatross*, and *black-footed albatross* represent subconcepts of *bird* as shown by the directed arcs. The labels with no surround are roles and the labels with rectangular surrounds are *individuals* which, in this case, are acting as values for the roles. The interpretation is that an observed *individual* of type *bird* may be recognized as a *tubenose* if it is observed that its *nostrils* are *external-tubular*, it *lives at sea*, and its *bill* is *hooked*. The class of the *bird* may be further refined by further observation as pictured. If a *bird* is classed as the concept *black-footed albatross*, then a rule is fired (*rule1* in the barred rectangle) which assigns "Black-footed Albatross" to its *name* role. A complete KDraw knowledge base would have information about many other birds, descriptions of the types of things that can play the value part for each role, and various other pieces of information such as the mutual exclusion among certain concepts.

2.3.2 Conceptual Graphs

Another form of knowledge representation is Conceptual Graphs (Sowa 1984). Conceptual graphs were developed by Sowa based on Peirce's work (see page 21) on existential graphs between 1897 and 1906. Conceptual Graphs were *originally* developed as graphs, although a text format (called "linear form") has evolved for ease of use by computer programs. Computerized graphical editors now exist for Conceptual Graphs, for example the *GrIT* (Ellis & Levinson 1992) system which provides a graphical interface to the PEIRCE system, which is a Conceptual Graphs database and inference engine.

Conceptual graphs are capable of cleanly representing an extremely wide range of knowledge forms – from predicate logic to natural language. For example, Figure 6 is a conceptual graph representation of the surprisingly complex sentence "Tom believes that Mary wants to marry a sailor." Conceptual graphs are bipartite, with squared boxes representing concepts (and individuals) and rounded boxes representing relations between the concepts (and individuals). The larger boxes in Figure 6 are also concepts but are a special kind of concept called contexts. The outer concepts may be read as "There is an

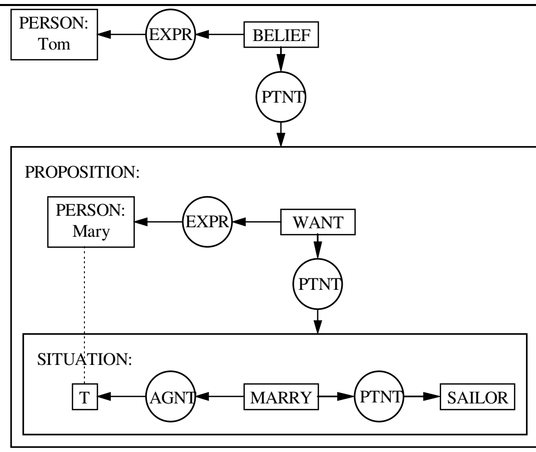


Figure 6: A concept map in the form of Sowa's Conceptual Graphs describing the sentence "Tom believes that Mary wants to marry a sailor." (Sowa 1984)

instance of a belief whose experiencer is the person Tom, and whose patient is a certain proposition." The proposition is "There is an instance of a want whose experiencer is a person Mary and whose patient is a certain situation." The situation is "There is an instance of marriage whose agent is something (T) and whose patient is some sailor." There is also a coreference link (the dotted line) between the concept [Person: Mary] and the concept [T]. This indicates that [Person: Mary] and [T] are the same individual, which modifies the interpretation of the situation to be "There is an instance of marriage whose agent is a person Mary and whose patient is some sailor."

The linear form of the conceptual graph in Figure 6 is

```
[BELIEF] -
 (EXPR) -> [PERSON:Tom]
 (PTNT) -> [PROPOSITION:
             [WANT] -
               (EXPR) -> [PERSON:Mary]
               (PTNT) -> [SITUATION:
                            [MARRY] -
                              (AGNT) -> [T:Mary]
                              (PTNT) -> [SAILOR]
                         ]
           ]
```

which represents the boxes (concepts) in Figure 6 with square brackets and the circles (relations) with round parenthesis. The arcs have fairly obvious translations using "->" and "<-". The one element that does not translate in a straightforward "graphical" manner is the coreference arc (dotted arc); coreference is represented only implicitly by the coincidence of the variable name *Mary*.

The above example serves to show that Conceptual Graphs can represent natural language sentences, but Sowa (1984) shows how Conceptual Graphs can be used to interpret natural language and to generate natural language as well. Furthermore, Conceptual Graphs are shown to represent first order logic (with an elegant set of visual deduction rules) and model logic.

The description of Conceptual Graphs given so far involves only the graphs themselves. A complete Conceptual Graphs definition is a *canon* consisting of the graphs, a type lattice (semantic net), the conformity relations (instance-of relations), and the formation rules. The graphs have already been discussed. The type lattice is a semantic net describing the subtype/supertype relationships among the concept and relation types used in the graphs. The conformity relations list describe the types of all individuals used in the graphs, so are ordered pairs from individuals to the elements in the type lattice. Finally, the formation rules are:

- *Restrict*: specializes the type of concept
- *Copy*: duplicates a graph.
- *Simplify*: removes any redundant (identical) relations between two concepts.
- *Join*: merges two graphs by unifying an identical concept node from each graph into a single concept.

Other operators (such as *maximal join* – joining two graphs on as many concepts as possible) are implemented in terms of these fundamental rules.

There now exist several tools for the storage and manipulation of Conceptual Graphs. These include PEIRCE (Ellis & Levinson 1992), Deakin ToolSet (Lukose 1991), and CGKEE (Munday et al. 1996). These tools are all based on the Conceptual Graphs linear form.

In addition, there have been several extensions to Conceptual Graphs. Lukose has developed Model-ECS (Lukose 1993; Lukose 1995) which is an executable form of Conceptual Graphs.

2.4 Other Concept Mapping Languages

Although visual knowledge representation languages are of primary interest in this thesis, other types of concept mapping languages are of interest as well. For example, gIBIS (Conklin & Begeman 1987) is a simple semi-formal language used in decision making and represents issues, positions, and arguments. There are also a vast number of visual programming languages which are, by definition, quite formal. These include OPAL (Musem, Fagen & Shortliffe 1987) and Pygmalion (Smith 1977) and many others (Myers 1990). Closely related to visual programming languages are program visualization languages (Price, Baecker & Small 1993) and languages used to design or represent programs. These include traditional flowcharting languages, and the newer object oriented programming notations such as Booch Notation (Booch 1994), Rumbaugh Notation/OMT Notation (Rumbaugh et al. 1991), Coad Notation (Coad, North & Mayfield 1995), and the more recent "Unified Method" (Booch & Rumbaugh 1995).

2.4.1 gIBIS

Figure 7 shows a concept map that may be developed by the participants in a decision-making meeting. The visual language used in Figure 7 is a semi-formalism called gIBIS (Graphical Issue Based Information System) (Conklin & Begeman 1987), which is an easy-to-use semi-formal language used for business decision making. Here, the elliptical

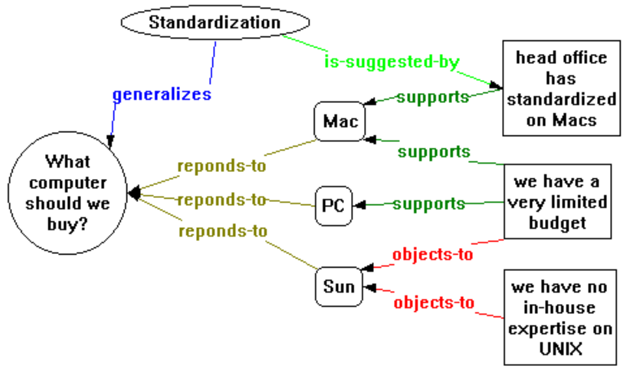


Figure 7: A concept map used to support a decision about buying a computer

nodes represent *issues*, the rounded-rectangular nodes represent *positions* taken on the issues, and the rectangular nodes represent *arguments*. The arcs connecting the nodes are all labeled with relationship types.

The decision making process represented in Figure 7 shows a clear and clean structure that would be quite difficult to represent using plain text. The relative linearity of text would obscure the relationships among the individual arguments and the sub-set of positions they relate to. To be sure, a two-dimensional table could clearly represent the structure of most of Figure 7 (with the exception of the "standardization" part), but tables quickly become difficult as the complexity of the situation increases. The flexible, two-dimensional structure of the concept map is much more capable of clearly representing complex knowledge structures.

2.5 Summary

This chapter has described visual languages as distinct from natural languages (such as English) and other textual languages (such as traditional programming languages (like C or Pascal). Visual languages use pictures to convey meaning in a two- or three-dimensional space. Visual languages range from very informal and unstructured to very formal and constrained. The value of visual languages is evidenced by the wide variety and large number of visual languages in existence, as well as by a number of compelling psychological arguments and some empirical evidence.

The syntax and semantics of visual languages is as varied as visual languages themselves, but Crimi et al (1991) have applied relational grammars to visual languages to formalize their syntax. Furthermore, Wang and Lee (1993) have addressed the semantics of visual languages by formalizing a system to describe an *interpretation* from the graphical (visual) domain to an application domain.

There exist several taxonomies of visual languages; however, Myers (1990) is the most well known and most inclusive. Myers uses three separate taxonomies to classify languages as programming systems, by specification technique, and by what artifacts they represent.

Concept maps are a general kind of visual language and are used for many purposes. Concept maps are graphs with nodes, arcs that interconnect the nodes, and possibly contexts which partition the map. Concept maps may be described in terms of Crimi et al's relational grammar and in terms of Wang and Lee's semantic interpretations, but *concept map* is too general a term to describe specific interpretations: interpretations must be applied to specific applications of concept maps.

Three specific concept mapping languages are described. Gaines' KDraw is an visual version of the CLASSIC knowledge representation language used in artificial intelligence applications. Sowa's Conceptual Graphs is also a knowledge representation language, and is used in natural language interpretation, natural language generation, and reasoning. Finally, a very simple decision-support language, gIBIS, is described.

Chapter 3

Background: Types, Objects, and Specification

This chapter deals with the object oriented paradigm and specification. There is dual motivation for describing the object oriented paradigm: The graphical meta-language developed in this thesis is an object-based system, and the language implementation and design use the object oriented paradigm.

Several issues in the object oriented languages design space are covered as background to issues that arise later in the development of Constraint Graphs. These issues include the classification of object oriented languages into class-based and object-based languages, several different inheritance models, and covariance, contravariance, and invariance. Furthermore, concept maps, which have already been discussed in chapter 2 are revisited in this chapter in light of the inheritance and typing issues discussed here.

Finally, specification and the Z specification language are briefly introduced as background to chapter 5.

3.1 Types, Classes, and Objects in the Object Oriented Paradigm

The object oriented paradigm is an important part of computer programming. To quote Abadi and Cardelli (1996) discussing the various programming paradigms:

> "…the object-oriented approach has proven uniquely successful. It manages to integrate analysis, design, and implementation techniques into a relatively intuitive and uniform framework." (p. 8)

There are a very large number of object oriented programming languages in existence today, with varying degrees of acceptance and use. There are several schools of thought on what constitutes an object oriented language; many of these schools would deny many of the others the label “object oriented”. For example, the “strongly typed” school might deny that (untyped) Smalltalk is an object-oriented language at all. Most of the differences pivot on the use and the conceptualization of objects, classes, and types.

What is the difference between an object, a class and a type? There are many definitions. Thought on the subject goes back to at least Plato. However, more recently, in philosophy, Zalta (1988; 1993) defines *abstract objects* (read "types") as *encoding* properties, while *ordinary objects* (read "objects") as *exemplifying* properties. Zalta's theory defines as many abstract objects as there are combinations (power sets) of properties. Thus, there are no two abstract objects (types) with exactly the same properties ($\exists!x(A!x \wedge \forall F(xF \equiv \varphi))$[1], where φ has no free xs), but two ordinary objects (objects) with exactly the same properties may be distinct. The set of all abstract objects (types) and the set of all ordinary objects (objects) cover the universe ($\forall x(O!x \vee A!x)$) and do not intersect ($\neg\exists x(O!x \wedge A!x)$).

Cardelli and Wegner (1985) view types as sets of values in their typed lambda calculus language *fun*. Cardelli and Mitchell (1994) define objects and classes in this way:

> Object-oriented programming is based on record structures (called objects) intended as named collections of values (attributes) and functions (methods). Collections of objects form classes. A subclass relation is defined on classes with the intention that methods work 'appropriately' on all members belonging to the subclasses of a given class. (p. 295)

In object oriented programming, Booch (1994) defines type, object and class:

---

[1] Zalta's notation warrants explanation: besides the usual quantifiers, the symbol $\exists!$ is the unique existential quantifier. Not to be confused with the quantifiers are the symbols A! and O! which represent predicates for "abstract type" and "ordinary type", respectively. In addition predicates over abstract types are written with the arguments first (e.g. xF), as opposed to predicates over ordinary types, which are written in the normal way, arguments last (e.g. Fx).

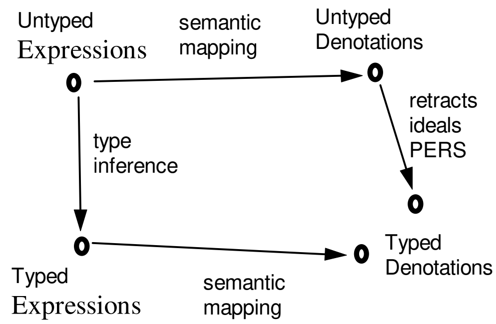


Figure 8: Type inference of the denotation or the expression do not necessarily arrive at the same conclusion (Wegner, 1988).

> Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways. (p. 66)
>
> A class is a set of objects that share a common structure and a common behavior. (p.103)
>
> An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable. (p. 83)

Hailpern and Nguyen (1987) list several meanings for objects and classes: *object* is merely a new name for *abstract data type* (ADT); *objects* and *classes* are a concrete form of type theory; *objects* and *classes* are a way of organizing and sharing code.

Although these definitions are different, there is some common thread throughout. Wegner (1988) sorts it out by subdividing the subject into four separate perspectives: classification, inheritance, mathematical, and philosophical:

- **The classification perspective** in biology and mathematics sees classes as equivalence classes (or relations). The kinds of equivalence relations are transformational equivalence, denotational equivalence, and type equivalence, where transformational equivalence ≤ denotational equivalence ≤ type equivalence.

Formalisms may infer the type of an expression either by determining the denotation and then determining the type of the denotation, or by inferring the type in the typed lambda calculus. These two methods do not necessarily arrive at the same conclusion (see Figure 8).

Classification in logic and computing can be by inference and unification, or by abstraction and specification. Classification in programming languages is one of three paradigms: state transition (statements), communication (objects), and classification (classes). Classification is complicated by polymorphism in its various forms.

- **The inheritance perspective** concerns itself with inheritance in object oriented programming languages. Object-oriented programming languages can be divided into three categories, *object-based* languages, which have only objects; *class-based* languages, which have objects and classes; and *object-oriented* languages, which have objects, classes, and inheritance. The combination of the first and last category, object-based inheritance, is possible and is exemplified by delegation. Specification of inheritance can be at three levels: signature (syntactic interface specification), types (implementation-independent behavior specification), and classes (implementation dependent specification by code). Variations on inheritance include nonstrict (behaviorally incompatible) inheritance, classless instance inheritance (delegation), abstract interface inheritance, and multiple inheritance.

- **The mathematical perspective** holds that types should be modeled by "establishing that algebras specify behavior while calculi specify rules for computing" (p.536). Types may be modeled by many-sorted algebras and subtypes may be modeled as relations on algebras.

- **The philosophical perspective** holds that the definition of *type* is different depending on the perspective (system evolution, system programming, type-checking, type inference, application programming, verification, implementation) taken. The nature of computational reality can be examined from the point of view of two philosophical dichotomies: objective (absolute) vs. subjective (relative), and realism vs. intuitionism.

The inheritance perspective is of particular importance in this thesis. Cook, Hill, and Canning (1994) distinguish between inheritance and subtyping and identify three types of inheritance: object inheritance (similar to C++'s inheritance), class inheritance (similar to Smalltalk's metaclasses), and type inheritance. But Cook's "object inheritance" is quite different from the inheritance in Wegner's object-based languages with inheritance. This alternate model of inheritance is called *delegation* and is important to this thesis in the Chapter 4. Delegation replaces the concept of a class with a *prototype* (Lieberman 1986). Any object can act as a prototype. When an object receives a message for which it does not have a method, it *delegates* responsibility to its prototype. Multiple inheritance in delegate languages is made possible by maintaining a list of prototypes for each object. Lieberman shows that delegation is more powerful than inheritance by showing that delegation implements the functionality of inheritance, but inheritance does not implement all the functionality of delegation.

It should be clear that there are no clear and absolute definitions of, or distinctions between, the terms *type*, *class*, and *object*. Class may not even be considered a primitive definition at all, as classes may be derived from objects (Abadi & Cardelli 1996). It all depends on one's perspective. It is clear, however, that types provide intentional definitions for objects and that objects represent instances (that may exist in some real or hypothetical world) which may be categorized according to type. The definition of class is far less clear: according to the programming language, class is the same as type (as in C++), or class is just an vehicle for ad-hoc inheritance and has nothing to do with type (Wegner 1988) (as in Java). Some new languages do not even use classes, but use prototypes instead and are still well-typed (Abadi & Cardelli 1996).

3.1.1 Object Oriented Programming Languages

Object oriented programming languages are represented by both paradigm-specific languages (Simula (Birtwistle et al. 1979), Beta (Kristensen, Madsen & Moller-Pdersen 1988), Smalltalk (Goldberg 1983)) and languages that have evolved from languages in other programming paradigms (C++ (Stroustrup 1991), Modula-2 (Wirth 1988)). *Object-oriented* is an over-applied term; many "object-oriented" languages fail to provide all the properties (objects, classes and inheritance) generally agreed upon as minimally defining "object-orientedness". Wegner (1988) provides a classification of near-object-oriented languages:

objects =	object-based	(Ada,Actors)
+ classes =	class-based	(Clu)
+ class inheritance =	object-oriented	(Simula, Smalltalk, C++)

There are many differences among "object-oriented" languages; these differences correspond to topics enjoying ongoing debate in the object-oriented programming community:

- Typing. Object-oriented languages may be untyped (Smalltalk), declaratively typed (Simula, C++), or inferentially typed (object-oriented extensions to ML (Remy 1994)) (Gunter & Mitchel 1994).
- Type inheritance vs. code-sharing inheritance. Some languages do not enforce child classes to be subtypes of the parent class; this non-strict inheritance replaces the *isa* relation with the weaker *like* relation (Wegner 1988).
- Class-inheritance vs. prototyping. Languages without classes can use delegation in place of class-inheritance. Delegation can be more powerful than inheritance (Lieberman 1986).

- Dynamic vs. static typing. Statically typed languages do all type checking at compile time (but languages with inheritance polymorphism allow type-safe dynamic (downward) resolution of *method* implementations).
- Concession to users "breaking" the type system. Some languages allow users to override the type system, such as C++'s cast operator.
- Single vs. multiple inheritance. The debate about whether the complexity of multiple inheritance is worth the effort continues to thrive.
- Exposure of instance variables. Classes have two types of clients for which they often want to provide different interfaces: the users of the class's objects, and the subclasses of the class. While it is generally accepted that classes should not provide outside users with direct access to instance variables (although many languages allow it), it is debated whether or not such direct access should be granted to the subclasses (Snyder 1988).

Abadi and Cardelli (1996) take a very wide view in their analysis and formalization of object oriented languages, encompassing typed and untyped languages, class-based and object-based languages, and embedding and delegation languages. They start off with a simple, untyped, non-inheriting record calculus. Objects are considered primitive and consist of a collection of methods (fields are modeled as just methods with no parameters – which implies that the language has method updates). From this simple calculus, they show how classes and inheritance can be modeled in terms of objects. Thus, they show clearly that objects can be taken as the primitive of object oriented programming.

Abadi and Cardelli model almost all the object oriented programming languages using their simple object calculi. Object oriented languages may be divided into class-based languages and object-based languages. In this model (unlike Wegner's), inheritance can occur in object-based languages. In class-based languages, classes act as templates for instantiating objects; in object-based languages, *prototypes* (concrete and fully functional objects) take the role of classes, and objects may be instantiated from them by *cloning*.

The inheritance roles in object-based languages are termed the *donor* (the "parent") and the *host* (the "child"). Object-based languages may be divided according to how they incorporate attributes of the donor object into the host object – either *implicitly* or *explicitly*, or, orthogonally, *embedded* or *delegated*. Implicit inheritance incorporates all attributes of the donor into the host, while explicit inheritance incorporates only specified attributes (but some languages may allow inheritance of groups of attributes, which implies that implicit/explicit inheritance is a continuum, not a bipolar quality). In embedded inheritance the donor attributes are copied into the host (although optimizations may blur this model); in delegate inheritance the donor attributes remain with the donor and the host accesses them as needed via a reference. Multiple inheritance is easily implemented by having each object maintain a list of donors.

Classes can be modeled on top of object-based languages merely by making host objects non-concrete, and differentiating the inheritance relationship between *isa* and *instance-of* for sub-classing and object instantiation respectively. Furthermore, *interfaces* (as per Objective-C and Java (Arnold & Gosling 1996)) are easily modeled as class-objects whose attributes include only method descriptions and no fields.

Abadi and Cardelli (1996) also make a contribution by clearly elucidating the constraints on method overriding and fields (attributes) in subtypes. It is a consequence of subsumption ($(a{:}A \wedge A \leq B) \Rightarrow a{:}B$) that method arguments are contravariant and return types are covariant. That is, any method that overrides a method of a parent type may generalize (but not specialize) the arguments and specialize (but not generalize) the return type. Furthermore, it is also a consequence of subsumption that record (or object) fields are always invariant. That is, a subtype may not change the type (generalize or specialize) of a supertype's field that may be publicly updated. In spite of this, some languages (for example, Eiffel (Meyer 1992) and Beta (Kristensen et al. 1988)) still allow covariance in method argument types. Beta uses run-time checks to catch errors resulting from covariant arguments.

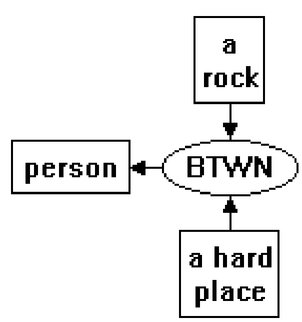
3.2 Types and Concept Maps

Although many concept mapping languages are not programming languages per se, their typing can be understood with reference to the object oriented paradigm.

3.2.1 Concept Maps as Hypergraphs

Concept maps are graphs, in the general graph theory sense, where nodes and arcs may be labeled and annotated with a type. Furthermore, if one does not make the restriction that arcs must be binary, but allows arcs to have arbitrary arity, then concept maps are *hypergraphs*. Hypergraphs are just graphs where the arcs can terminate on any number of nodes (Drewes & Kreowski 1990). A graph (in the graph theory sense) is just a specialization of a hypergraph where every arc (or edge) is binary (has an arity of 2).

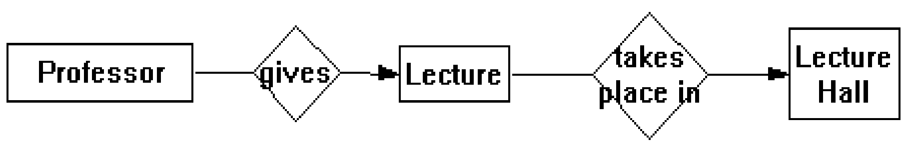
If concept maps are to be modeled using either Crimi's relational grammars or Wang and Lee's semantic interpretations, then the obvious choice is to allow concept maps to be hypergraphs, since neither Crimi's nor Wang and Lee's formalisms assume binary relations (see Section 2.1.2). Furthermore, generality alone argues for a hypergraph interpretation. There is also the compelling argument that some situations are difficult to model without non-binary relations, for example, *between* (Sowa 1984):



Here, BTWN is just the label on a trinary arc. It can be argued that non-binary arcs can be replaced by nodes with special restrictions (for example, the trinary *between* arc in the preceding example could be replaced by a node which is restricted to have one incoming and two outgoing arcs). But such a model can often lead to complex semantic restrictions. The more general, hypergraph, interpretation is taken as the superior because such a concept mapping system can subsume both *between* models.

3.2.2 Concept Maps and Relations

If one models general concept maps as hypergraphs, it becomes easy to map between concept maps and relational structures just as Crimi maps between general visual language and relations to apply relational grammars, and Courcelle (1990) maps between hypergraphs and relational structures. Corrandini et al (1994) show how labeled graph grammars can be used to specify entity-relationship diagrams where the type of the arcs (relations) restricts the node (entity) types on which the arcs may terminate. Thus, the definition graph,



will restrict an arc (relation) of type *gives* to connecting only a node (entity) of type *Professor* to one of type *Lecture*, and will restrict an arc (relation) of type *takes place in* to connecting only a node (entity) of type *Lecture* to one of type *Lecture Hall*. Corrandini et al work only with graphs (and not hypergraphs), but the principle projects to hypergraphs in a straightforward manner. As a case in point, Crimi's relational grammar for visual syntax is very similar, but works with hypergraphs (see Section 2.1.2).

Both Corrandini et al and Crimi's work is applied to *labeled* (hyper)graphs. But it is a simple matter to extend this work to deal with *typed* hypergraphs. Here, *labeled* hypergraph is taken to be a special kind of typed hypergraph where the types of nodes and

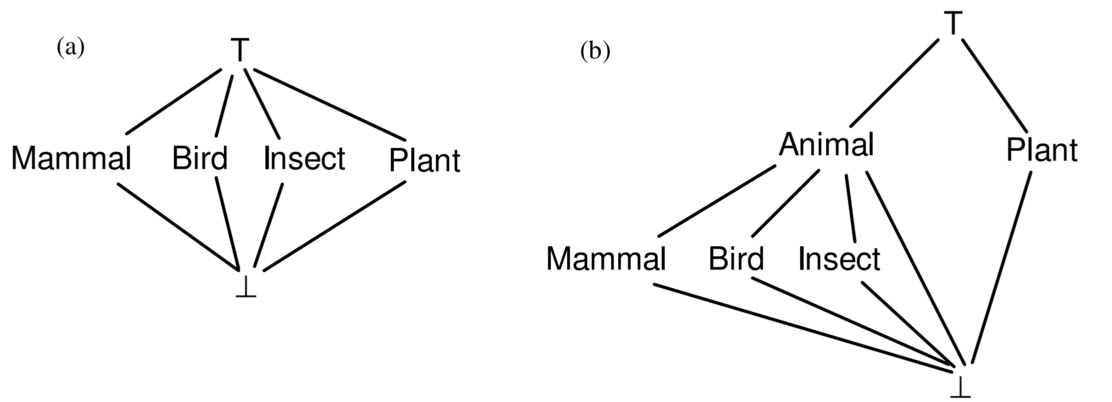
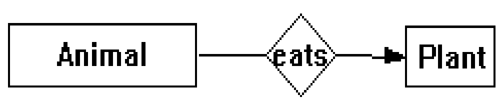


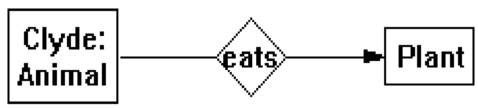
Figure 9: An example of (a) a flat type lattice as might be used in a *labeled* hypergraph, and (b) a type lattice as might be used in a *typed* hypergraph

arcs are taken from a flat type lattice: there in no subtype/supertype relationship between the different types (other than the universal and absurd types). In contrast, in a *typed* hypergraph the types of nodes and arcs are taken from an arbitrary type lattice: there exists a subtype/supertype relationship between the different types (see Figure 9).

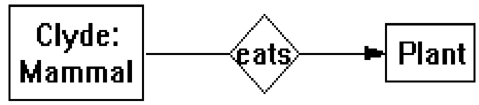
Typed hypergraphs make Corrandini's arc constraints slightly more complex because subsumption must be taken into account. Take, for example, the following definition graph under the typing of Figure 9 (b):



In a flat-typed (labeled) world, this definition allows an object of type animal to eat a plant:



but it wouldn't allow a more specific (specialized) animal to eat the same plant:



because *Mammal* isn't the same label as *Animal*. Clearly, knowing more about Clyde's type shouldn't necessarily bar him from eating. In a typed hypergraph world Clyde can eat: Informally, the type of the node at an arc terminal is legal if and only if it is some subtype of the type of the node at the corresponding terminal in the arc's definition. In other words, it's legal to put a *Mammal* node at the start of an *eats* arc because *Animal* subsumes *Mammal*.

But there can be a type lattice for arcs as well as nodes (for example, *eats* may be a subtype of *consumes*, which may be a subtype of *effects*). Therefore, the reference to an arc's "definition" in the previous paragraph needs to be replaced with a reference to an arc's "supertype": The type of the node at an arc terminal is legal if and only if it is some subtype of the type of the node at the corresponding terminal in all of the arc's supertypes.

In more formal terms, one needs to discuss instances and types separately. If one is discussing instances, the arcs can terminate on subtypes of the definition-allowed nodes due to subsumption (e.g. $(\text{Mammal} <: \text{Animal} \wedge \text{Clyde:Mammal}) \Rightarrow \text{Clyde:Animal}$). If one is discussing types, arcs are covariant in the node types that they may terminate on. This is easy to see by noticing that the type of an arc may be considered to be the product of the types of the nodes on which it can legally terminate. For example, if binary arcs of type A are restricted to terminate on nodes of types X and Y, then $A =_{def} X \times Y$. Abadi and Cardelli (1996) offer a proof that product types are covariant:

> A pair <a,b> with left component a of type A and right component b of type B, has type A×B. If A <: A' and B <: B', then by subsumption we have a:A' and b:B' , so that <a,b> has also type A'×B'. Therefore any pair of type A×B has also type A'×B' whenever A<:B' and B<:B'. In other words, the inclusion A×B <: A'×B' between product types is valid whenever A<:A' and B<:B'. (p. 20)

Since arc types are product types, and product types are covariant, then arc types are covariant on their "arguments" (the types of the nodes on which they may legally terminate).

Thus, the relational grammar approach of Crimi, and the labeled graph grammar approach of Corrandini et al can be applied to typed hypergraphs (and concept maps in particular) with only a small addition to account for subsumption and covariance.

3.3 Formal Specification and Z

Software formal specification has spread from the academic environment to industry. Craigen, Gerhart, and Ralston have conducted detailed studies of twelve large industrial software development projects in commercial, exploratory, and regulatory domains (Craigen, Gerhart & Ralston 1993; Gerhart, Craigen & Ralston 1994). They concluded that, though formal methods are still not mature, they "are beginning to be used seriously and successfully by industry to design and develop computer systems" (p. ii). Formal methods are being applied to large-scale projects, and are used primarily for system assurance, domain analysis, stakeholder communication, evidence of best practice, re-engineering, and system certification. While tool support was found to be lacking, Craigen, Gerhart, and Ralston (1993) found tool support to be "neither necessary nor sufficient for successful application of formal methods" (p. 70); but rather, successful application depends on local skills, sustained guidance by an expert, management support, and technological change providing an impetus for adoption of formal methods.

Many of Craigen, Gerhart, and Ralston's case studies used Z or Z-derived methods as the formal specification notation. Z is a relatively simple algebraic notation based on set theory which makes heavy use of schemata to block structure the specification. Z has been used to model a wide range of applications. Hayes' (1987) book contains case studies of Z specifications of the UNIX file system, telephone networks, data dictionaries, assembler languages, user name authentication, time services, reservation services, and various parts of the CICS operating system. Craig (1991) describes an extended Z specification of the blackboard architecture commonly used in AI applications, as well as the related CASSANDRA architecture. Halasz and Schwartz (1990) use Z to specify a hypertext architecture, which has been extensively adopted and extended (Gronbaek et al. 1994; Gronbaek & Trigg 1994; Hardman, Bulterman & Rossum 1994).

One of the attractive features of Z is its compatibility with the object-oriented paradigm (Craig 1991). Classes in object-oriented programming languages are based on block structure records (Madsen 1988) which maps directly onto Z's schemata. Classes form the types of object oriented programming languages, and objects are instantiated from classes – just as schemata form the types of Z and variables are instantiated from schemata. Furthermore, class inheritance is considered fundamental to object-oriented programming and design (Pree 1995; Kristensen, Madsen, Moller-Pedersen & Nygaard 1988; Snyder 1988; Wegner 1988) and is partially modeled in Z: schemata may be included in other schemata. This simple inclusion scheme allows Z to model both single and multiple inheritance. (In addition, schemata may be combined by various operators such as conjunction, disjunction, and sequential composition; this powerful tool has no mapping in the object-oriented paradigm.)

3.4 Summary

This chapter has dealt with the object oriented paradigm and its relevance to concept maps. It also briefly introduced specification and the Z specification language.

Pivotal to the object oriented paradigm are the notions of type, class, and object. Unfortunately, there are no clear definitions or sharp distinctions between these terms. Types intentionally describe objects, which are more-or-less concrete instances. But the definition of class is more complex, sometimes being taken as the same as type, sometimes being taken as just an implementation vehicle. Both the notion of types and classes is clouded by the notion of meta-types and meta-classes.

Several relevant issues in object oriented languages are discussed. These will be of importance in chapter 4. There are many classifications of object oriented languages: typed vs. untyped, type inheritance vs. code-sharing inheritance, class-inheritance vs. prototyping, and dynamic vs. static typing to name just a few. Of particular interest is class-inheritance vs. prototyping: prototyping blurs the distinction between type (or class) and object because a fully-functional object can act as a "type" or class when an instance is *cloned* from it. Prototype languages with no classes may also use inheritance in two ways: by embedding (copying attributes) and delegation (reference to the original donor).

Concept maps can be modeled as hypergraphs, where the arcs need not be binary, but may be of arbitrary arity. In this model, concept map arcs can be mapped onto relations and can be dealt with using the semantics of relational grammars (Courcelle 1990) and hypergraph grammars (Corrandini et al. 1994). Using these grammars, arcs can be easily constrained as to the type of the nodes their terminals may be anchored to. This work all deals with "flat" type lattices for node and arc types, but it is easily extended to an arbitrary type lattice model by using subsumption and Abadi and Cardelli's work on covariance in object oriented language theory.

The role of specification is also briefly outlined. Specification and specification tools have moved from academia to industry. Z is a popular and simple specification language based on set theory. A dialect of Z is used to specify Constraint Graphs in chapter 5.

Chapter 4

Requirements Analysis

This chapter describes the objectives and requirements for the Constraint Graphs visual meta-language system. The system is intended to be a platform in which one can easily prototype a wide variety of visual languages, and visual knowledge representation languages in particular. The generic visual language type is called a “concept map”, which includes a wide variety of visual languages.

Although the presentation of the requirements in this chapter makes it appear that the requirements were developed before the specification and software were begun, this is not the case. Like most software development efforts, the detailed design, implementation, and testing stages uncovered further requirements that were not apparent during the initial analysis. For example, until it was found that *individuals'* visual attribute inheritance from *concepts* lead to some unexpected properties in implementing the KDraw language, the designer was unaware of the need for multiple type lattices. At that point the requirements were revisited to add **Requirement 18** to the list of requirements: the new requirement was woven into the specification, and the new feature was added to the implementation.

The requirements were first derived by studying several concept mapping languages – KDraw and Conceptual Graphs were first studied before the specification and implementation were begun. Once the first working implementation was produced, Constraint Graphs was tested by applying it to the definition of other languages such as gIBIS and several object oriented design languages. Although these tests produced no significant changes to the requirements (probably owing to their relative simplicity compared to knowledge representation languages), more detailed implementations of the original two languages, KDraw and Conceptual Graphs, produced some very interesting results. For example, a reformulation of the implementation of Conceptual Graphs to take advantage of arc constraints could yield a much more powerful system with considerable semantics checking (see Section 7.2). In addition, the post-implementation detailed implementation of KDraw, adding visual attribute inheritance from concepts, produced some unexpected new requirements as mentioned above.

There is no reason to believe that the set of requirements listed here will remain static. But rather, it is expected that the requirements (and the theory) should evolve dynamically over time – as more and more languages are implemented under Constraint Graphs.

4.1 Objectives

4.1.1 A Framework for Visual Knowledge Representation

The basic objective of this work is to develop a concept map framework under which one may easily build various formal and informal visual knowledge representation notations.

Objective 1	Develop a general and flexible concept map framework for visual knowledge representation.

Ideally, one should be able to create new graphical knowledge representation languages in a recursive fashion by using concept maps themselves to specify the features in the new language. An obvious corollary is that the framework should be able to imitate existing knowledge representation languages, and this is where most of the emphasis will be placed, since it avoids the trap of creating a language with features specific to a particular model. Examples of knowledge representation languages to be captured by this

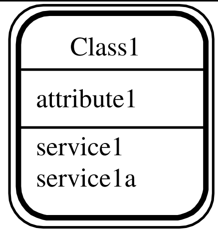


Figure 10: A typical node from Object Model Notation

framework are Gaines' KDraw (Gaines 1991b) and Sowa's Conceptual Graphs (Sowa 1984).

Since this objective emphasizes generality and flexibility, it is also appropriate to choose examples of graphical, semi-formal, non-knowledge representation languages to generalize the test applications. Examples of such languages are gIBIS (Conklin & Begeman 1987) for decision support and Object Model Notation (Coad, North & Mayfield 1995) for object-oriented program design.

The objective does *not* imply that the implementation is intended to *exactly* match the target visual languages. For example, Object Model Notation uses nodes with three distinct sections (see Figure 10); for the purposes of this implementation it will be considered adequate to simplify by avoiding the double border and use semicolons in place of the separator lines.

On the other hand, certain natural extensions to existing notations should be considered appropriate for the implementation. For example, Conceptual Graphs rely on an associated semantic net, composed in a separate notation, to specify the type lattice. The examples in Sowa's 1984 book (Sowa 1984) use the textual notation in the appendix as their associated semantic net. For the purposes of this work, it is a reasonable (and compelling) extension to allow both the Conceptual Graph and semantic net to be combined in the same graph.

4.1.2 Formal Basis

Since this work is intended to be a framework used as the base for other systems and as a model for the theory, the structure of its design must be very clear and unambiguous. One appropriate path to this end is to use formal methods, since formal methods will provide a logically precise design and serve as a reference point to resolve ambiguities. However, software engineering principles dictate that wholesale application of formal methods may be inappropriate since over-application of formal methods will increase the required effort of the project beyond reasonable levels (DeMarco 1982). The "engine" component of the framework is deemed appropriate for formal specification, while the user interface is viewed as an add-on (replaceable) component that would be inappropriate to formally specify. It is the engine component of the framework that models the theory and needs to be unambiguous and rigorous if it is to show its accurate modeling of the theory.

Objective 2	The design should be formal and rigorous where appropriate.

4.1.3 Principled Development

Formal methods will be used for the core parts of this work that are most subject to ambiguous interpretation. This does not imply that the rest of the work should be done haphazardly: the entire work, from design to coding should be done a principled way. Both the parts that are formally specified and the parts that are not should be properly designed, documented, and coded taking advantage of standard class libraries and techniques. A clear and principled design and development process will contribute to the confidence in the accurate correspondence between the software framework and the theory.

Objective 3	The target system should be developed in a principled manner.

A recent approach to principled software development is the explicit use of *design patterns* (Gamma et al. 1995; Pree 1995), which provide detailed, semi-formal documentation of design concepts that have proven track records in past projects. One of the advantages of design patterns is that they provide a vocabulary and starting point for design, and many of them can be encoded as general-purpose classes in object-oriented class libraries.

In addition, this work should take full advantage of existing principled tools. For example, the C++ Standard Template Library (STL) (Nelson 1995; Stepanov & Lee 1995), is a container library which has been developed and documented in a principled way.

4.2 Overall Requirements

The first requirement is that there must be a clear separation between the part of program that enforces the rules of layered visual languages (the constraint graph) and the part that implements the visual graphics and accepts user commands (the user interface).

Requirement 1	The system should have a clear separation between the constraint graph engine and the user interface.

This requirement addresses **Objective 1** (flexibility and generality) by allowing configurations with differing constraint graph engines and user interface engines. Such a design yields high flexibility and generality. A monolithic system may not be sufficiently flexible. This requirement also supports **Objective 3** (principled development).

As an example, one might consider the choice of implementation languages. The constraint graph system and the user interface system differ in their computation attribute considerably: The constraint graph sub-system is likely be to quite computationally complex (since it is known many type systems have polynomial or exponential complexity), while the user interface sub-system (graphical drawing) is not likely to be a computational bottleneck because the reasonably simple drawing operations are very fast compared to the interaction rate of the human user. Furthermore, the constraint graph sub-system, since it has nothing to do with the user interface, is likely to be easily portable between operating systems (if written in a reasonably portable language), while the user-interface is not likely to be easily portable between operating systems because its many graphical input/output operations will be implemented quite differently on the different operating systems. The two parts of the program differ in computational complexity and in ease of porting to another operating system. C or C++ would be a reasonable language choice to address the computational complexity of constraint graph sub-system, but, on the other hand, the limited portability of user interface libraries does not address the concerns of the user interface sub-system. Java is a cross-platform portable language, which addresses the user interface sub-system, but, on the other hand, it is relatively slow, and will not perform well as the language of the constraint graph sub-system. Since the two sub-systems differ remarkably in their feature profiles, it may make good sense to implement them in different languages. Only a clean separation between the two will yield the flexibility to do this.

4.3 Constraint Graph Requirements

A Constraint Graph is defined here as a very abstract graph, in the graph-theoretic sense, that contains the usual nodes and arcs. In addition, it enforces a strict typing on the nodes as defined by a special arc type, called *isa*. Several other forms of constraints are also applied, as detailed below, in order to satisfy **Objective 1**, a framework in which to emulate and experiment with visual knowledge representation systems.

The Constraint Graph itself will have no user interface of its own (see **Requirement 1**). However it *will* have an interface to other programs that can implement an interface on its behalf.

4.3.1 Same Interface for Ontology and Domain Notation

Many graphical knowledge representation systems, whether formal or informal, use separate and/or non-graphical notational mechanisms to represent the type lattice of graphical components. Two examples are Accord (Kremer 1993), which uses a complex dialog box to edit its type hierarchy; and Conceptual Graphs (Sowa 1984), which uses a separate and distinct semantic net to present its type lattice. This is not unexpected: there is a firm distinction between types and objects. In philosophy, Zalta (1988; 1993) defines *abstract objects* (types) as *encoding* properties and *ordinary objects* (objects) as *exemplifying* properties. But there are many definitions of types and objects (see Section 3.1), depending on one's perspective.

On the other hand, other languages such as CLASSIC (Borgida et al. 1989) and KDraw (Gaines 1991b) cleanly integrate the ontology (type relationships) graph with the domain graph. This makes for a clean and clear representation of knowledge without forcing the user to learn several different tools or sub-formalisms.

Constraint Graphs should cleanly integrate ontology and domain data in a single graph.

Requirement 2	Ontology graphs should be cleanly integrated with domain graphs.

This integration is necessary to support integrated languages like KDraw. Furthermore, it does no damage in representing languages like Conceptual Graphs because the ontology is a clean "add-on" that can be ignored by traditionalists. **Requirement 2** supports **Objective 1**.

In the case of Constraint Graphs, where the objectives include both flexibility and rigor, one must be very careful. In many cases, the user will require Zalta's rigorous distinction between type (astract objects) and objects (ordinary objects); but in other cases this rigor may be damaging to the flexible support of multiple visual languages. For example, one may build up layers of ontologies upon one another. It is not always clear to the user which is a type and which is an object; it depends on the user's perspective. For example, one may have a "universal" ontology (representing "thing", "physical_thing", "intangible_thing", etc.), and use the universal ontology as the basis for an ontology of machines, and use the machine ontology as the basis for an ontology about photocopy machines, and use the photocopy machine ontology as the basis for an ontology about the Xerox-9000 line of photocopy machines, and finally use the Xerox-9000 ontology as the basis for a domain graph about specific photocopy machines on the third floor of the library. Each successive ontology behaves as though it were a domain based on the ontology above it. Furthermore, although the chain of ontologies appears to terminate with the specific machines, this is not necessarily the case. One could use the specific photocopy machine domain as an ontology for a domain about the machine variants caused by engineering updates to the machines (where the type changes over time, but not identity).

Another case where the distinction of type and object becomes blurred is in dealing with metaclasses. A class defines the properties of an object, but a metaclass defines the properties of a class. For example, the metaclass might define a superclass, a name identifier, a list of attributes, and a list of methods.

The point is that Constraint Graphs needs to use a typing system with a great deal of flexibility: there isn't always a clear distinction between type/class/concept and instance/object/individual, and some entities may have to viewed sometimes as an type and sometimes as an instance.

This notion of not clearly recognizing the distinction between type and instance may seem somewhat heretical, but it is not without support in other systems. Object-based languages do not contain the notion of class, but many of these languages can support a form of inheritance through prototypes (fully functional objects used to "template" new objects) and either embedding or delegation (see Section 3.1.1). Abadi and Cardelli (1996) make a very good argument that "objects are more primitive than classes and should be understood and explained before classes" (p. 49) and that classes arise out of objects. Furthermore Lieberman (1986) shows that delegation is more powerful than class-based inheritance.

It therefore makes good sense in Constraint Graphs to refuse to recognize as primitive the differences between type and instance:

Requirement 3	The system should enforce no intrinsic distinction between types and instances.

If a system needs to distinguish between types and instances (as in KDraw's *concepts* and *individuals*) one can do it by tagging a parent type of the "instances" with a constraint (see **Requirement 15**) such as "no subtype may itself have any subtypes"

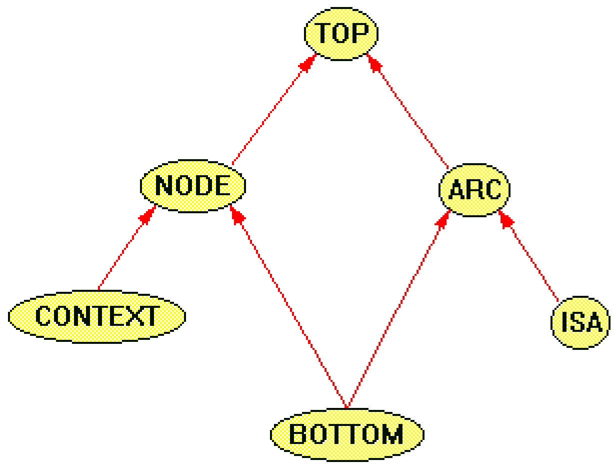


Figure 11: The base type lattice for Constraint Graphs

4.3.2 Maximize Simplicity / Minimize System Types

A good rule-of-thumb for any software engineering project (and for any undertaking whatsoever) is "keep it simple". It follows that the base type system for Constraint Graphs should be as simple and succinct as possible.

4.3.2.1 The base type lattice

Graph theory dictates that at least a node type and an arc type are required. Node and arc are mutually exclusive. Since type theory is also supported, a means to represent the type hierarchy is also required. This can easily be accomplished by using a specialized directed arc type called *isa*, which is, itself, a subtype of arc.

In addition, some mechanism for grouping and nesting is required. Conceptual Graphs uses *contexts* to group sub-graphs into "possible worlds". Examples of Conceptual Graphs contexts are NEGATION, PROPOSITION, and SCENE. Contexts are strictly nested, and so form a hierarchy of sub-graphs. While contexts are not used in CLASSIC, they are used in KDraw (which is based on CLASSIC) as a useful visual aid to the user (there is no associated semantics). Since contexts are used exactly as nodes in Conceptual Graphs, contexts should be a subtype of the node type. One could argue that contexts are related to arcs since they "connect" or group other objects, but this argument is not sound because arcs form ordered collections and have no hierarchy, while contexts form unordered collections and are hierarchical (see Section 4.3.2.4). The base type lattice for Constraint Graphs is given in Figure 11.

Requirement 4	The base type lattice should be "context < node < top > arc > isa", where *node* and *arc* are mutually exclusive.

4.3.2.2 Arcs

In mapping from Conceptual Graphs to Constraint Graphs, Conceptual Graphs' relations map to Constraint Graphs' arcs quite naturally. (One could also map relations onto nodes and make special bipartite rules.) Since Conceptual Graphs relations are of arbitrary arity (but not zero), Constraint Graphs is compelled to make arcs have arbitrary arity. See also Section 3.2.1.

Requirement 5	Arcs are n-ary, but must have arity greater than zero.

While binary arcs are by far the most common in visual languages, Sowa (1984) makes compelling arguments to support n-ary relations. For example the *between* relation is very hard to represent using only binary arcs. Ordered lists can also be conveniently represented as open-ended n-ary arcs.

Many visual languages contain directed as well as undirected arcs. Both types can be easily simulated by keeping the terminals of arcs ordered.

Requirement 6	Arc's terminals are ordered.

Many visual languages are "first-order" (drawing a parallel to predicate calculus) in that they allow arcs only between nodes (and not other arcs). But some visual languages are "higher-order" in that they *do* allow arcs between arcs. This dictates that Constraint Graphs must support arcs between arcs.

Requirement 7	Arcs may terminate on objects of type node or arc.

This feature is also used to form a type lattice of arcs (**Requirement 9**) which serves to restrict arc terminals (**Requirement 14**).

4.3.2.3 *Isa* arcs

It has already been established that *isa* is a subtype of arc. But the semantics of *isa* still needs to be specified. *Isa* arcs are directed, binary arcs. (The "directed" part comes for free since arc terminals are ordered as per **Requirement 6**.) *Isa* arcs are constrained to be binary since there is no semantic difference between a component inheriting from two supercomponents via a single (trinary) *isa* arc, and inheriting from the same two supercomponents via two distinct (binary) *isa* arcs. Allowing higher-arity *isa* arcs only introduces unnecessary redundant methods and complicates specification and implementations.

Requirement 8	*Isa* arcs are strictly binary.

One should be able to just "draw" in an *isa* arc between two objects to establish the subtype/supertype relationship between them. The type system should work on the basis of *isa* arcs, and either accept or reject modifications based on the lattice formed by the *isa* arcs (including the insertion and deletion of *isa* arc).

Requirement 9	The subtype/supertype relationship is established by drawing *isa* arcs between objects. The type system accepts or rejects graph modification based on the lattice of *isa* arcs within the graph.

4.3.2.4 Contexts

Contexts are specializations of nodes that "contain" subsets of the graph.

Requirement 10	Contexts are a subtype of node and so behave in the same manner as nodes. Arcs can terminate on contexts, as well as on objects contained within contexts.

Contexts constitute partitions on a graph; that is, contexts represent subsets of the components of a graph. N-ary arcs may also be considered as representing graph partitions, but one must be careful to distinguish between n-ary arcs and contexts: contexts are set-like, so their referents are *unordered* with respect to the context; n-ary arcs are sequence-like, so their referents are *ordered* with respect to the arc.

Requirement 11	Contexts contain sub-graphs of the graph; the components of the sub-graph are unordered with respect to the context.

Contexts form a strict hierarchy of subsets of a graph; they cannot overlap. This is in concurrence with Sowa's Conceptual Graphs. Logically, contexts differ from arcs in that they form a hierarchy, and objects they "refer to" form an unordered collection.

Requirement 12	Contexts are strictly nested: they form a hierarchy.

One could argue that there is no inherent reason to make contexts hierarchical; after all, one of the objectives is to make the system as flexible as possible. It might make sense to allow partially overlapping contexts, then constrain partial overlaps away when necessary (as in the case of modeling Conceptual Graphs). Unfortunately, when one starts considering the physical layout of context boxes that contain subsets of each others' graphs, things quickly get out of hand. Two context boxes sharing a common sub-graph is not very difficult to draw, but the problem gets exponentially more difficult with more context boxes sharing arbitrary sub-graphs. For example, Vein diagrams work well for two or three sets, but the geometry gets increasingly difficult when one adds more sets. Even certain combinations of five sets are impossible to draw using contiguous two-dimensional shapes.

This Pandora's box is best avoided by simply conforming to Conceptual Graphs' hierarchical contexts. When sub-graphs must be shared between contexts, it can be done using coreference arcs as per Sowa's Conceptual Graphs.

4.3.3 Maximize Flexibility

In support of **Objective 1**, the design of Constraint Graphs must strive to be as flexible and powerful as possible in order to support representation of a large number of visual languages. It should also be easy to use.

Requirement 13	The graphical syntax of common visual languages should be implementable in the system without excessive low-level programming.

The next two requirements are corollaries of **Requirement 13**.

4.3.3.1 Graph syntax specification

The basic syntax of concept mapping languages is primarily concentrated on restricting the types of arcs that may be drawn between objects based on the objects' type. This can be accomplished naturally (and graphically) by use of the subtype relationship among arcs. As show in Section 3.2.2, arc types are covariant on the types of the nodes on which they may legally terminate. That is, any subtype arc may specialize the legal node type on which any of its terminals anchor. This may be reworded the other way around as **Requirement 14**:

Requirement 14	An arc may terminate only on objects that are subtypes of the corresponding terminal in all the arc's supertypes.

For example, in order to restrict the *mother* relationship arc to be between objects of *person* and type *female-person*, one only need create a prototype arc (from type *person* to *female-person*) and derive all *mother* arcs from this prototype. Figure 12 shows a simple ontology/domain describing the concepts *person, female-person*, and *male-person* together with the individuals *Sally* and *Joe*. *Isa* arcs are shown as unlabeled, directed arcs. A labeled binary directed arc is drawn from *person* to *female-person* (which is the prototype *mother* arc) to restrict the type of the terminal objects. Finally, a new arc of type *mother* (also labeled *mother*) is drawn between Joe and Sally. This arc is legal because Joe *isa person* and Sally *isa female-person*. The arc could not be drawn in the other direction because Joe does not have type *female-person*.

A problem may arise in the situation in Figure 12. Note that **Requirement 14** talks about *corresponding* arc terminals, but does not restrict the *addition* of arc terminals (i.e., increasing the arity of an arc) in a subtype. This is as it should be. But occasionally one *may* want to restrict the arity of all subtypes of an arc. In the case of *mother*, this probably makes sense.

It turns out there are a large variety of restrictions that are necessary to emulate other visual languages. Some examples are:

- Conceptual Graphs has quite a bit of syntax associated with the concept labels.
- KDraw carries syntax on the label of its constraint nodes.

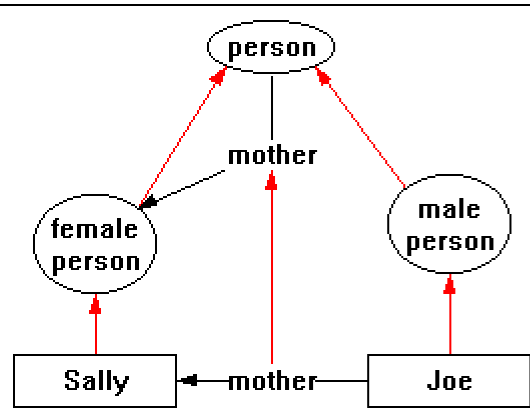


Figure 12: Restricting the *mother* relationship terminals

- KDraw requires prevention of multiple inheritance through certain types.
- KDraw requires objects of type *Individual* (but not *Individual* itself) to *not* have any subtypes.

To enumerate all of these possibilities is probably not possible for all conceivable visual languages. It makes sense to allow some mechanism for the user to easily extend the system to handle these cases.

Requirement 15	Some mechanism must exist to allow the user to "program" arbitrary constraints. These ad-hoc constraints could be collected in a library for easy use.

4.3.4 Strong Typing

The type system should be maintained at all times. That is, modifications to the graph should be disallowed if they violate constraints.

Requirement 16	Strong typing – all objects and types should conform to all their, and their supertypes' constraints.

This means that whenever any object in the graph is modified, created, or deleted, the entire graph must be checked for constraint violations. In practice the entire graph does not have to be checked since there is a limit to the effect of a change. Changes will only affect objects below the changed object in the type lattice (subtypes), but all constraints of objects above each affected object (every supertype) will have to be checked against the affected objects. In addition, object modification will force constraint checks in their adjacent objects (for changed arcs, terminal objects; for all objects, the arcs that terminate on them). **Requirement 15** (arbitrary constraints) can cause this constraint checking scheme to fail.

4.3.5 Incompleteness – Dangling Arcs

During the development of visual graphs there are always intervals where some part is inconsistent due to incompleteness. For example, while drawing a binary arc in single-step is an easy task (usually elastic line dragging), it is not so easy to draw a trinary arc in a single step. Instead, it makes more sense to allow the creation of the trinary arc with "dangling" terminals, and let the user "attach" the terminals as a separate operation.

The system must make some trade-off between always maintaining completeness and usability. Some constructs are not possible without temporary incompleteness or incorrectness. In general, simply allowing for dangling arc terminals is sufficient.

Requirement 17	The system should allow for incompleteness (e.g. dangling arcs) when the graph is in an intermediate state.

Dangling arc terminals should always be considered a stand-in for an object of the correct type. In this respect, dangling arc terminals are analogous to null pointers in programming languages.

4.3.6 Multiple Type Lattices

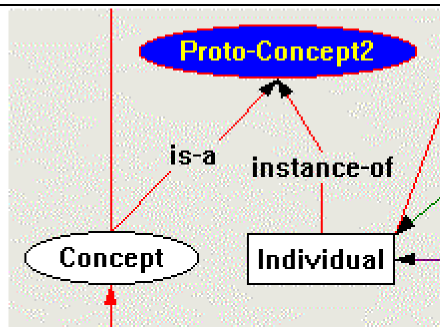


Figure 13: A fragment of a KDraw specification using Constraint Graphs

An interesting problem arrises when mapping KDraw to Constraint Graphs. In KDraw and CLASSIC (Borgida et al. 1989) a distinction is made between *concepts* and *individuals* (unlike Constraint Graphs). The fact that *individuals* can't have subtypes is easily handled by a constraint (**Requirement 15**). But both the subtype (*concept*-to-*concept*) and instance-of (*individual*-to-*concept*) arcs seemed best modeled as subtypes of the base *isa* arc used by the Constraint Graphs' type system (see Figure 13). This allows an object of type *individual* to inherit attributes and constraints from its supertype *concepts*, as it should. But it also causes the individual to inherit properties (such as being the root of a *has-rule* arc) that violate the syntax of KDraw. Clearly, this indicates two separate (but closely related) type lattices superimposed on the graph. One of these lattices is the KDraw syntax and the other is the domain model. The two are exactly identical except that the *instance-of* arc is an *isa* arc in the domain model lattice, but not in KDraw syntax lattice.

Requirement 18	The system needs to support overlayed (but distinct) type lattices. *Isa* arcs can be tagged with the set of type lattices to which they apply.

KDraw's *individual/concept* problem is solved using a filter on the subtypes of *instance-of* arcs to distinguish the two separate lattices. *Isa* arcs are tagged with the set of type lattices to which they apply. The type inference system can use the KDraw syntax ontology filter (which won't see the *isa* arc between an *individual* and a *concept*), while the attribute resolution system uses the domain model filter (which does see the *isa* arc between an *individual* and a *concept*).

Most *isa* arcs are tagged with the universal set.

4.3.7 Attributes

In the interests of supporting an independent user interface (**Requirement 1**), the Constraint Graphs system must be able to cleanly carry information on behalf of the user interface. This information should be associated with the objects (nodes and arcs) in the graph. Furthermore, this information should be inherited by subtypes of objects. In short, the objects in the constraint graph need inheritable attributes with values. Inheritance is standard with almost any object oriented system, but specifically, this system uses *delegate* inheritance (see Section 3.1.1) which means that not only attributes and their types are inherited, but so are their values because the slot for the attribute is stored not in the host (subtype) but in the delegate (supertype). If attribute is not found in the host, it is searched for in the delegate(s). This is similar to inheriting a "default" value with the attribute, but the "default" can change if its changed in the parent.

Requirement 19	All objects in the graph will have associated attribute/value pairs that are inherited (delegated) through the type lattice.

Delegation of attributes does not imply that an attribute value cannot be overridden. This is normally done by placing a slot in the host (subtype), thereby avoiding searching for the attribute in the delegate (supertype).

Requirement 20	Attributes and their values are inherited, but may be overridden.

Although attribute *values* may be overridden, the attribute *type* may not be overridden or specialized in a subtype. Attributes that can be updated turn out to be invariant (neither covariant or contravariant). Abadi and Cardelli (1996) offer a proof based on pairs whose components can be updated (operator "♦"):

> If A<:A' and B<:B', can we covariantly allow A♦B <: A'♦B'? If we adopt this inclusion, then from p:A♦B we obtain p: A'♦B' and we can perform setLft(p,a') for any a':A'. After that, getLft(p) might return an element of type A' that is not an element of type A. Hence the inclusion A♦B <: A'♦B' is not sound.
>
> Conversely if A"<:A and B"<:B, can we contravariantly allow A♦B<:A"♦B"? From p:A♦B we now obtain p:A"♦B", and we can incorrectly deduce that getLft(p):A". Hence the inclusion A♦B <: A"♦B" is not sound either. (p. 21)

Requirement 21 is therefore necessary:

Requirement 21	Attribute values should be typed, and the attribute value type cannot be overridden by a subtype.

It is also sometimes necessary to prevent a subtype from changing the value of an attribute. For example, in KDraw, a concept is always represented by an elliptical surround; to override this attribute would render the graph unreadable. Preventing overriding is analogous to the *const* tag in C and C++.

Requirement 22	It should be possible to prevent overriding of an attribute's value.

4.3.8 Ontological Levels

As discussed in detail for **Requirement 2**, there are usually several ontological levels; this is particularly true for as system such as Constraint Graphs where one has a base level (arcs and nodes), then defines a formalism (such as KDraw or Conceptual Graphs), then defines an ontology (or several levels of ontology), then defines a domain. It makes good sense to be able to group all the objects into their respective levels.

Requirement 23	Graph objects should be partitioned into ontological levels.

The levels can then be used for various purposes. For example, the end user (the one who works on the bottom-level domain) does not often want to be concerned with anything but the lowest-level ontology. Therefore, choices the end user makes about type can be restricted to just that specific ontology level. Likewise, inferences the system makes about possible object typings can be restricted (and often completely disambiguated) by searching only in restricted levels.

4.3.9 Levels of Formalization

Although the constraint graphs system is intended to be a formal system, it is also intended to be a useful and usable system. Usefulness and usability imply that the system must be able to handle situations in which users find it inconvenient or impossible to enter complete and correct graphs (Kremer 1994; 1995).

Requirement 24	Support for varying levels of formalization.

While **Requirement 17** addresses incompleteness, **Requirement 24** is a more general statement which also encompasses incorrectness. One possibility for addressing this requirement is to allow the constraints (**Requirement 15**) to be tagged with an urgency value, which would allow the type system to violate constraints if the user so desired.

4.4 User Interface Requirements

The user interface is not a central focus of this thesis, but it is critical to the usefulness and usability of this system. This section therefore briefly describes the requirements for a user interface to be used with the Constraint Graph system.

Besides the obvious requirement of being a flexible and extensible system that can call into the Constraint Graph system and receive update messages from it, the interface elements must conform to following list of requirements.

4.4.1 Nodes

Requirement 25	Interface nodes should be labeled or unlabeled with little or no restriction on the characters in the label.

Requirement 26	Interface nodes should have a reasonably rich set of visual attributes, including, but not restricted to, node shape, fill color, border color, and text color

4.4.2 Arcs

Requirement 27	Interface arcs must be able to be visually anchored to nodes, contexts and other arcs and to track them when they are moved.

Requirement 28	Interface arcs should be labeled or unlabeled with little or no restriction on the characters in the label.

Requirement 29	Interface arcs should have a reasonably rich set of visual attributes, including, but not restricted to, label shape, fill color, border color, and text color; line color; arrowhead style and color.

Requirement 30	Interface arcs should have arbitrary arity. They must also have some method of visually distinguishing the order of the terminals.

4.4.3 Contexts

Requirement 31	Interface contexts should be labeled or unlabeled with little or no restriction on the characters in the label. They must be capable of visually containing subsets of the graph.

Requirement 32	Interface contexts should have a reasonably rich set of visual attributes, including, but not restricted to fill color, border color, and text color.

Requirement 33	There should be some mechanism in the interface were contexts may be restricted to be independent or nested, but not partially overlapped.

4.5 Summary

This chapter has detailed the objectives and requirements of the Constraint Graph system. The primary objective is to produce a powerful and flexible system in which one can describe and implement a variety of visual language formalisms. The complete list of objectives is listed below:

Objective 1	Develop a general and flexible concept map framework for visual knowledge representation.
Objective 2	The design should be formal and rigorous where appropriate.
Objective 3	The target system should be developed in a principled manner.

The complete list of Constraint Graph requirements is listed below:

Requirement 1	The system should have a clear separation between the constraint graph engine and the user interface.
Requirement 2	Ontology graphs should be cleanly integrated with domain graphs.
Requirement 3	The system should enforce no intrinsic distinction between types and instances.

Requirement 4	The base type lattice should be "context < node < top > arc > isa", where *node* and *arc* are mutually exclusive.
Requirement 5	Arcs are n-ary, but must have arity greater than zero.
Requirement 6	Arc's terminals are ordered.
Requirement 7	Arcs may terminate on objects of type node or arc.
Requirement 8	*Isa* arcs are strictly binary.
Requirement 9	The subtype/supertype relationship is established by drawing *isa* arcs between objects. The type system accepts or rejects graph modification based on the lattice of *isa* arcs within the graph.
Requirement 10	Contexts are a subtype of node and so behave in the same manner as nodes. Arcs can terminate on contexts, as well as on objects contained within contexts.
Requirement 11	Contexts contain sub-graphs of the graph; the components of the sub-graph are unordered with respect to the context.
Requirement 12	Contexts are strictly nested: they form a hierarchy.
Requirement 13	The graphical syntax of common visual languages should be implementable in the system without excessive low-level programming.
Requirement 14	An arc may terminate only on objects that are subtypes of the corresponding terminal in all the arc's supertypes.
Requirement 15	Some mechanism must exist to allow the user to "program" arbitrary constraints. These ad-hoc constraints could be collected in a library for easy use.

Requirement 16	Strong typing – all objects and types should conform to all their, and their supertypes' constraints.
Requirement 17	The system should allow for incompleteness (e.g. dangling arcs) when the graph is in an intermediate state.
Requirement 18	The system needs to support overlayed (but distinct) type lattices. *Isa* arcs can be tagged with the set of type lattices to which they apply.
Requirement 19	All objects in the graph will have associated attribute/value pairs that are inherited (delegated) through the type lattice.
Requirement 20	Attributes and their values are inherited, but may be overridden.
Requirement 21	Attribute values should be typed, and the attribute value type cannot be overridden by a subtype.
Requirement 22	It should be possible to prevent overriding of an attribute's value.
Requirement 23	Graph objects should be partitioned into ontological levels.
Requirement 24	Support for varying levels of formalization.

The complete list of interface requirements is listed below:

Requirement 25	Interface nodes should be labeled or unlabeled with little or no restriction on the characters in the label.
Requirement 26	Interface nodes should have a reasonably rich set of visual attributes, including, but not restricted to, node shape, fill color, border color, and text color.

Requirement 27	Interface arcs must be able to be visually anchored to nodes, contexts and other arcs and to track them when they are moved.
Requirement 28	Interface arcs should be labeled or unlabeled with little or no restriction on the characters in the label.
Requirement 29	Interface arcs should have a reasonably rich set of visual attributes, including, but not restricted to label shape, fill color, border color, and text color; line color; arrow head style and color.
Requirement 30	Interface arcs should have arbitrary arity. They must also have some method of visual distinguishing the order of the terminals.
Requirement 31	Interface contexts should be labeled or unlabeled with little or no restriction on the characters in the label. They must be capable of visually containing subsets of the graph.
Requirement 32	Interface contexts should have a reasonably rich set of visual attributes, including, but not restricted to fill color, border color, and text color.
Requirement 33	There should be some mechanism in the interface were contexts may be restricted to be independent or nested, but not partially overlapped.

Chapter 5

Specification

This chapter describes the specification of the Constraint Graphs system based on the requirements listed in the previous chapter. The design is split very firmly between the semantic engine (or abstract constraint graph), which is the heart of Constraint Graphs, and the user interface as per **Requirement 1** (see Figure 16 on page 113). It is only the semantic engine (the Constraint Graph itself) that is specified here. There is no formal specification for the user interface, although it is described in the following chapter.

The Constraint Graphs system is specified in a dialect of the specification language Z (Hayes 1987; Spivey 1989) called ZSL (Jia 1995). Z was chosen primarily because it is simple (based on set theory) and popular. Since the implementation paradigm is object-oriented programming (the system is implemented in C++), one of the object-oriented enhancements of Z (Stepney, Barden & Cooper 1992a; 1992b) would have been more appropriate. For example, inheritance polymorphism is extremely awkward to express in plain Z. Unfortunately, a good type checker for object oriented Z notations was not available at reasonable cost at the time of specification.

ZTC (Jia 1995) is a type checker for Z that is available in the public domain. It can take several syntactic dialects of Z as input: two Latex versions, and a pure ASCII text version. The pure text version is particularly appealing as it can be easily incorporated in standard HTML documents. Due to a serendipitous coincidence of compatible syntax, ZTC will even type check an HTML document with embedded ZSL.

The following description of the specification is in bottom-up order. This is necessary for a type checker, since terms need to be defined before they are used. Unfortunately, this is rather awkward for people. Nonetheless, the description will follow the bottom-up order necessitated by the Z type checker. To mitigate this problem, an overview (Section 5.1) is provided to introduce the specification.

A more serious problem is that of modeling inheritance in Z. When modeling the supertype/subtype relationship, the definition of the supertype is usually given first. This can be done in Z by including the supertype schema in the subtype's schema definition. But since Z is not object oriented, it does not embrace inheritance. Thus, if there is a schema A, the definitions

```
---B--          ---C--
| A             | A
------          ------
```

do not imply B<A (B is a subtype of A) or C<A. Furthermore, values of type B and C cannot be referred to using the common type. To get around this problem, inheritance is modeled in an "upside-down" way using free type definition. In the example above, A0, B0 and C0 are defined first as stand-ins for A, B, and C; these contain the definitions one would expect in A, B and C. Then a free type definition is used to capture a common reference for B0 and C0:

```
A_types ::= b << B0 >> | c << C0 >>
```

A can now be defined using A0 and a new attribute called Role which can reference objects of type B0 and C0:

```
---A-------
| A0;
| Role: A_types
|-----
| A0
-----------
```

Finally, B and C can be defined as just A's with a restriction over the Role attribute:

```
B =^= [A | Role in ran b]
C =^= [A | Role in ran c]
```

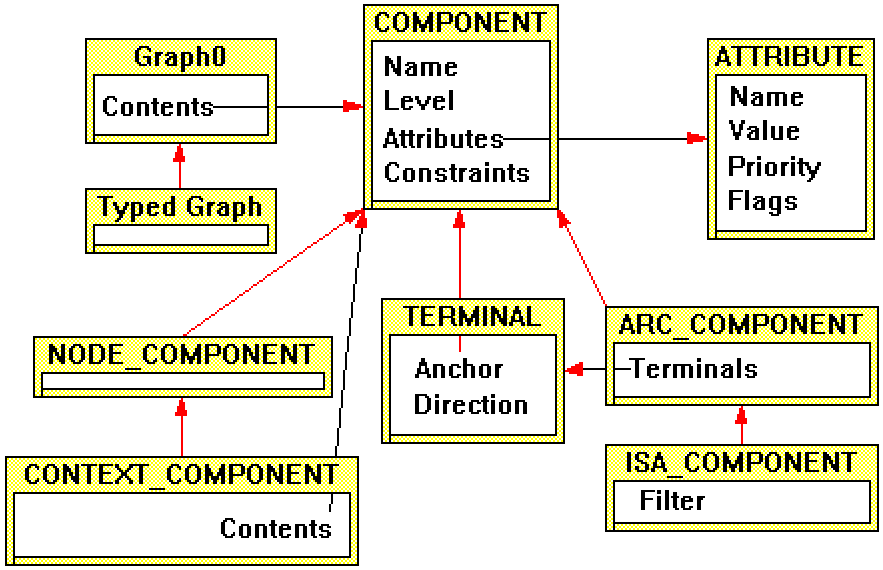
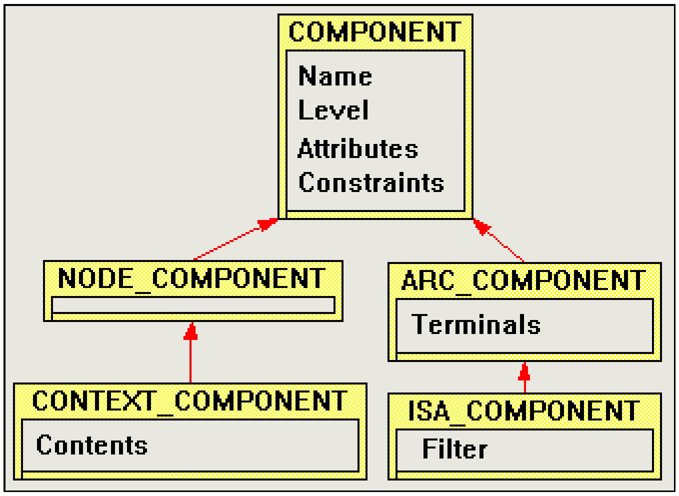


Figure 14: A simplified and idealized overview diagram of the Constraint Graphs specification

Although this technique is awkward and rather opaque, it does capture the typing that inheritance would yield (B and C are just specialized A's), and it passes the Z type checker just as though it where a true inheritance situation.

5.1 Overview of the Specification

Figure 14 is an idealized diagram of the Constraint Graphs specification presented in this chapter. It is idealized because it shows an inheritance structure that is not possible to model directly in the Z specification language. Nonetheless, Figure 14 is very useful to explain the structure of the specification.



The five component "schemata" form a lattice almost identical to that of specified by **Requirement 4** and diagrammed in Figure 11 on page 64 (the only thing missing is the specific disjoint between the node and the arc types).

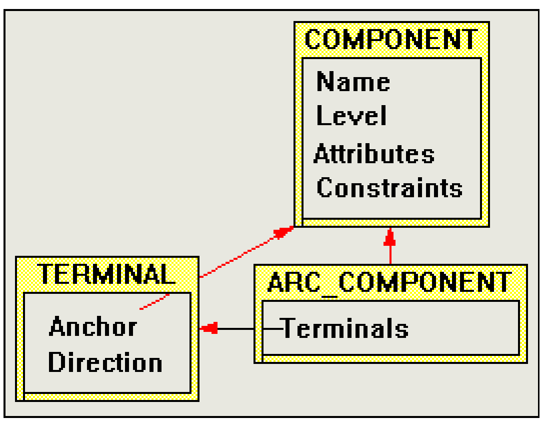
COMPONENT is specified as having a Name, a Level (**Requirement 23**), Attributes (**Requirement 19**) and Constraints (**Requirement 15**). Names support human usability and aren't really used for anything in the specification or the implementation, except to make things easier for the user to understand. Levels divide the components of graphs into categories. In general, lower-numbered levels contain the "primitive" components while successively higher-numbered levels contain progressively higher-order components. Levels are useful for several purposes, such as:

- A layered system (such as a user interface) may need to infer possible component types (such as the type of an arc between two nodes or the possibilities for a new node type). By restricting the "new" components to be from a certain set of levels, the visual language designer can limit the search space to a reasonable subset of the world.
- A visual language designer may want to hide certain constructs from an end user (such as low-level ontological details). This may be done simply by placing the hidden constructs in a separate level, and hiding the level.
- A visual language designer may want to lock out the end user from changing certain "fundamental" constructs in order to prevent the end user from disrupting the integrity of the visual formalism. This can be done by placing the locked components above a certain level, and preventing any edits above that level.

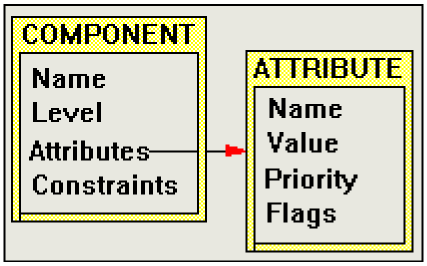
5.1.1 Nodes

NODE_COMPONENT specializes COMPONENT, but does not particularly extend it. NODE_COMPONENT is nonetheless essential because COMPONENT is a "virtual" type which is never directly subclassed (except type NODE_COMPONENT and ARC_COMPONENT). The rule that NODE_COMPONENT and ARC_COMPONENT are mutually exclusive could not be enforced without NODE_COMPONENT. CONTEXT is a simple extension to NODE_COMPONENT which contains a subgraph of nested components. In fact, Figure 14 itself is drawn using Constraint Graphs, and the labeled boxes are CONTEXT_COMPONENTS that contain subgraphs: in this case, just simple NODE_COMPONENTS (with no shape surround).

5.1.2 Arcs

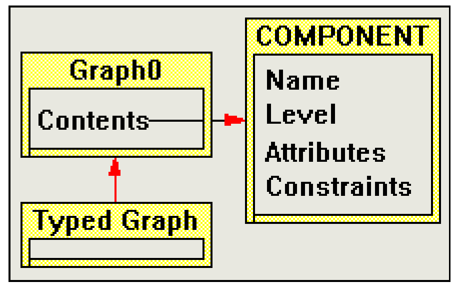


ARC_COMPONENT specializes COMPONENT and extends it with a sequence of Terminals which can, together, be physically represented by n-ary connection lines between components. All the arcs in Figure 14 are binary arcs, but higher-order arcs could be represented by forked lines of various sorts. ISA_COMPONENT specializes ARC_COMPONENT by adding an ontology filter in support of **Requirement 18**. ISA_COMPONENT also adds further constraints on *isa* arcs, such as "*isa* arcs are binary", which don't show up in the diagram but do show up in the specification.



Components have sets of attributes as per **Requirement 19** which consist of an attribute name, an arbitrary value of any type (but there are constraints to do with inheritance, **Requirement 21**), a priority to disambiguate in multiple inheritance situations, and some flags used to support **Requirement 22** (preventing overriding of attribute values).

5.1.3 Constraint Graphs



Graph0 is the basis of Constraint Graphs, and primarily acts as a container for the components. It also has responsibility to maintain a number of invariants among the contained components. TypedGraph is a specialization of Graph0 which adds to Graph0 the ability to enforce constraints in the components. Graph0 could not do this itself because constraints cannot be fully defined before Graph0 is defined.

Figure 15 shows a much more accurate depiction of the specification than does Figure 14. Since Z does not handle inheritance, it must be imitated in a rather opaque manner as described in the introduction to this chapter. Figure 15 shows the Z nesting (or inclusion) of specifications. The important inheritance relationship between COMPONENT, NODE_COMPONENT and ARC_COMPONENT has been turned into a COMPONENT with a free type definition, called "Role", which is defined as one (of any) of the subtypes in the Figure 14 model.

This section has provided an overview of the structure of the Constraint Graphs specification. The remainder of the chapter is devoted to the specification proper.

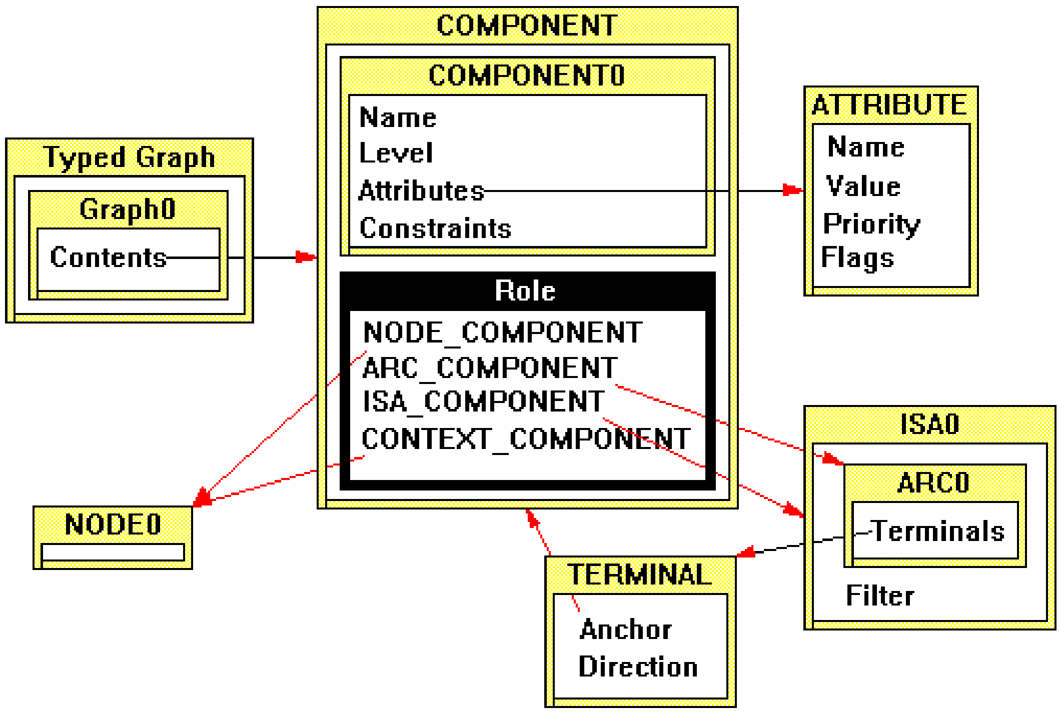


Figure 15: A simplified, but more accurate diagram of the Constraint Graphs specification

5.2 Attributes

ATTRIBUTEs are used to tag graph components with application-specific attributes (**Requirement 19**). For example, the attribute "Shape" (Name) might describe the surround of a graph node with values (Value) such as "rectangle" or "ellipse". Attributes are meant to be inherited through the type system. The Priority field is used to arbitrate in the case of conflicts arising from multiple inheritance of an attribute. The Flags field is used to tag attributes with special properties such as CONSTANT, which prevents subtypes from overriding the value of the attribute (**Requirement 22**), and PRIVATE, which prevents subtypes from being able to “see” the attribute.

```
[char, ATTRIBUTE_NAME]

ATTRIBUTE_VALUE ::= pos_int << N >>
          |    int << Z >>
          |    string << seq char >>

| attribute_value: ATTRIBUTE_VALUE % generic value

ATTR_FLAGS ::= CONSTANT | PRIVATE

---ATTRIBUTE------------------------------------------------------------
| Name: ATTRIBUTE_NAME;
| Value: ATTRIBUTE_VALUE;
| Priority: Z;
| Flags: P ATTR_FLAGS;
------------------------------------------------------------------------
```

The ATTRIBUTE_VALUE specification above is not intended to be an inclusive set of type possibilities: The implementation would be expected to extend the types as necessary.

MakeAttribute is a convenient function to create a new attribute.

```
| MakeAttribute: ATTRIBUTE_NAME & ATTRIBUTE_VALUE & Z & P ATTR_FLAGS
|                                          --> ATTRIBUTE
|-----------
| MakeAttribute =
|    (lambda n:ATTRIBUTE_NAME; v:ATTRIBUTE_VALUE; p:Z; f:P ATTR_FLAGS @
|      (mu a:ATTRIBUTE | a.Name=n /\
|                        a.Value=v /\
|                        a.Priority=p /\
|                        a.Flags=f))
```

5.3 Ontology Filters

Ontology filters support the ability to overlay multiple ontologies within a single graph (**Requirement 18**). Recall from **Requirement 9** that type lattices (or ontologies) are formed by *isa* arcs between the graph's components. By tagging *isa* arcs with information as to whether it applies to a particular ontology, one can examine any particular ontology by merely ignoring all the *isa* arcs that are not applicable. This is done with ontology filters.

To begin with, assume that each ontology has a unique identifier:

```
[ONTOLOGY_ID]
```

An ontology filter is then just a set of ontology identifiers:

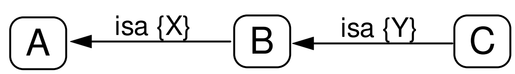
```
| OFilter: P (P ONTOLOGY_ID);
```

Thus, when one is examining a type lattice, one need only check to see if each of the defining *isa* arc's ontology filters contains the ontology identifier of the ontology (or ontologies) of interest. Note that there is no reason to focus on any particular ontology identifier, but rather a set of ontology identifiers may be compared against a filter.

The above description is a bit over-simplified. Quite often, one wants to check a component's ancestry over *any* ontology, so one uses the set of all ontologies (the *universal ontology*) as the ontology of interest.

```
UniversalOntology == ONTOLOGY_ID
```

This is simple enough, but it is possible that, even when using the universal ontology, there may be a chain of *isa* arcs that cannot be followed. For example, in the following diagram,



the *isa* arcs are labeled with the set of ontology identifiers, and X and Y are distinct ontology identifiers. Naively, one might assume that C<A (C is a subtype of A), but this is not so. Since X and Y are distinct, C is not related to A at all. Even when one starts out from C with the universal ontology as the ontology set, the ontology set should be restricted by the filter on the C→B *isa* arc,

$$\text{ontology} \cap \text{filter}(C \rightarrow B) = \text{UniversalOntology} \cap \{Y\} = \{Y\}$$

which causes the B→A *isa* arc to be ignored:

$$\{Y\} \cap \text{filter}(B \rightarrow A) = \{Y\} \cap \{X\} = \varnothing$$

The following relation captures this concept:

```
%% inrel accepts

| _ accepts _ : OFilter <-> OFilter
|-----------------------
| forall f,f2:OFilter @
|       (f accepts f2) <=> ((f && f2) /= {})
```

Both the set of interest ontologies and the filter are considered ontology filters.

5.4 Basic Components: Nodes and Arcs

COMPONENT0 is the common part of both the graph nodes and the arcs among the nodes. All components have:

- a name,
- a level (to be described later, **Requirement 23**),
- a set of ATTRIBUTEs (**Requirement 19**), and
- a set of VALIDATORs (**Requirement 15**).

```
[VALIDATOR, NAME]

---COMPONENT0---------------------------------------------------------
| Name: NAME;
| Level: Z;
| Attributes: P ATTRIBUTE;
| Constraints: P VALIDATOR;
----------------------------------------------------------------------
```

COMPONENT is a specialization of COMPONENT0 that is meant to be the supertype of nodes, arcs, *isa*-arcs, and contexts (**Requirement 4)**, but one needs to first define arcs, *isa*-arcs and contexts (since they all refer to COMPONENTs). Therefore COMPONENT is pre-declared here.

```
[COMPONENT]
```

CONTENTS is the contents of a node, which will be left up to the application and so is just defined generically. But CONTEXT will be a sub-type of node, and its contents will be a set of other components; so this is explicitly mentioned as a possible CONTENTS type.

```
CONTENTS ::= GENERIC_CONTENTS
          | members << P COMPONENT >>
```

NODE0 is the node-specific extension to COMPONENT0. It has some unspecified contents.

```
---NODE0-----------------------------------------------------------------
| Contents: CONTENTS;
-------------------------------------------------------------------------
```

Arcs take a little more work to specify than nodes. Arcs are very general in this specification: they have an arbitrary arity (they need not be binary, **Requirement 5**), are ordered (**Requirement 6**), and they may terminate at an arbitrary component (either nodes or other arcs, **Requirement 7**). The terminals of an arc have direction besides a reference to the anchor component. All arcs must have at least one terminal (**Requirement 5**).

```
DIRECTION ::= FROM | TO | BIDIRECT | NONE

---TERMINAL--------------------------------------------------------------
| Anchor: COMPONENT;
| Direction: DIRECTION;
-------------------------------------------------------------------------
```

ARC0 is the extension of COMPONENT0 for arcs.

```
---ARC0------------------------------------------------------------------
| Terminals: seq TERMINAL;
|---------------
| #Terminals > 0
-------------------------------------------------------------------------
```

5.5 Isa

Isa arcs are a subtype of arc with an ontology filter (**Requirement 18**) and a restriction that they be binary (**Requirement 8**).

```
---ISA0-----------------------------------------------------------------
| ARC0;
| Filter: OFilter;
|----------------
| #Terminals = 2
------------------------------------------------------------------------
```

5.6 Component

Finally, the parts can be brought together to define a component. The basic components of a graph are nodes, arcs, and *isa* arcs. A COMPONENT is the base, COMPONENT0 (name, level, and attributes) together with the NODE0, ARC0, or ISA0 extension.

```
COMPONENT_TYPES ::= node << NODE0 >>
           |    arc << ARC0 >>
           |    isa_arc << ISA0 >>

---COMPONENT-------------------------------------------------------------
| COMPONENT0;
| Role:COMPONENT_TYPES;
|-------------
| COMPONENT0;
-------------------------------------------------------------------------
```

NODE_COMPONENT, ARC_COMPONENT, ISA_COMPONENT, and CONTEXT_COMPONENT are simply conveniences to refer to arcs, nodes, *isa* arcs, and contexts. Contexts are just nodes that contain other graph components (**Requirement 10** and **Requirement 11**).

```
NODE_COMPONENT =^= [COMPONENT | Role in ran node]

ARC_COMPONENT =^= [COMPONENT | Role in ran arc]

ISA_COMPONENT =^= [COMPONENT | Role in ran isa_arc]

CONTEXT_COMPONENT =^= [COMPONENT | Role in ran node /\
                          (node~Role).Contents in ran members]
```

NULLCOMPONENT is a special value used to represent "dangling" arcs (**Requirement 17**).

```
| NULLCOMPONENT: COMPONENT
```

5.7 Some Convenient Functions

MakeTerminal is a convenience function to create a terminal:

```
 MakeTerminal: COMPONENT & DIRECTION --> TERMINAL
---------------
 MakeTerminal = (lambda c:COMPONENT; d:DIRECTION @
        (mu t:TERMINAL | t.Anchor=c /\ t.Direction=d))
```

The functions Terminal and TerminalDir are conveniences to extract individual terminals and direction of individual terminals from arcs.

```
 GetTerminal: ARC_COMPONENT & N +-> COMPONENT;
 GetTerminalDir: ARC_COMPONENT & N +-> DIRECTION;
--------------
 forall a:ARC_COMPONENT; n:N | n <= #(arc~(a.Role)).Terminals @
    GetTerminal(a,n) = (((arc~(a.Role)).Terminals)(n)).Anchor /\
    GetTerminalDir(a,n) = (((arc~(a.Role)).Terminals)(n)).Direction;
```

The GetFilter function is a convenience to extract the filter from an ISA_COMPONENT .

```
 GetFilter: ISA_COMPONENT --> OFilter;
--------------
 forall a:ISA_COMPONENT @
    GetFilter(a) = (isa_arc~(a.Role)).Filter;
```

The incontext relation is a convenience to determine whether or not a particular component is a member of a context.

```
%% inrel incontext
```

```
 _ incontext _ : COMPONENT <-> CONTEXT_COMPONENT
--------------
 forall c:COMPONENT; x:CONTEXT_COMPONENT @
   c incontext x <=> c in members~((node~(x.Role)).Contents)
```

5.8 A Basic Graph

So far, the discussion has centered around the parts of a simple graph; however, the intent is to describe typed graphs. Nodes and arcs all have one or more supertypes. In this specification, types are just other components of the graph (**Requirement 2** and **Requirement 3**). A type or object is a subtype or instance, respectively, of another by virtue of being connected to it by a special arc type called ISA_COMPONENT (**Requirement 9**).

The parts of a graph (nodes, arcs, *isa* arcs, and contexts) are themselves type objects in this theory. A node is called "Node", etc.

```
| Node,Arc,IsA,Context,T: NAME
```

The four components (plus the defining *isa* subtypes between them) all reside uniquely in the highest type level (level 1). A graph always contains the basic components: node, arc, context, and *isa*.

Within a graph, one can describe the traditional subtyping relations. Here the relation *parentof* refers to only immediate parents, while the relation *ancestorof* is the normal proper supertype/subtype relation. *Isa* is as expected: just the same as *ancestorof* except that identity is included. Note that the direction of the *isa* relation is the reverse of the other two!

```
%% inrel parentof
%% inrel ancestorof
%% inrel isa
```

There are several constraints on a GRAPH0:

- Node, Arc, *Isa*, and Context are the only level 1 objects.
- There are no cycles in the *isa* relation.
- Everything is uniquely either a node or an arc (which means nothing has an *isa* arc chain to more than one of the two basic types; nodes and arcs are mutually exclusive, **Requirement 4**).
- Contexts can only contain components that are part of the graph (**Requirement 11**).

In order to more conveniently talk about *isa* chains, it is necessary to define a *path*. A PATH is a sequence of COMPONENT pairs (together with the relevant filters) that leads from any COMPONENT to an ancestor.

```
PATH == seq ((COMPONENT & OFilter) & COMPONENT)
```

PATH is used in the definition of the function, *paths_btwn*, which will be used to examine ancestry relationships between components.

Finally, the definition of GRAPH0 follows. Note that this definition is incomplete because it does not include anything about constraints (which are called for by **Requirement 15**). This is because the definition of constraints (next section) requires the definition of a graph. The definition of a graph can be extended from GRAPH0 after constraints are defined in terms of GRAPH0.

```
---GRAPH0---------------------------------------------------------------
| Contents: P COMPONENT;
| TOP: COMPONENT;
| NODE: NODE_COMPONENT;
| ARC: ARC_COMPONENT;
| ISA: ISA_COMPONENT;
| CONTEXT: CONTEXT_COMPONENT;
|
| Level1Objects: P COMPONENT;
|
| _ parentof _ : COMPONENT & OFilter <-> COMPONENT;
| _ ancestorof _ : COMPONENT & OFilter <-> COMPONENT;
| _ isa _ : COMPONENT & OFilter <-> COMPONENT;
|
| % these next 3 awkwardly shadow the above due to syntax constraints
| parentof0 : COMPONENT & OFilter <-> COMPONENT;
| ancestorof0 : COMPONENT & OFilter <-> COMPONENT;
| isa0 : COMPONENT & OFilter <-> COMPONENT;
|
| paths_btwn : COMPONENT & COMPONENT & OFilter --> P PATH;
| get: N1 >+> Contents;
| getID: Contents --> N1;
|--------------
| getID = get~;
|
| {TOP, NODE, ARC, ISA, CONTEXT} = Level1Objects;
| forall c,p:Contents; o:OFilter @ (p->o)->c in parentof0   <=>
|                                              (p->o) parentof   c;
| forall c,p:Contents; o:OFilter @ (p->o)->c in ancestorof0 <=>
|                                              (p->o) ancestorof c;
| forall c,p:Contents; o:OFilter @ (c->o)->p in isa0         <=>
|                                              (c->o) isa    p;
| Level1Objects subseteq Contents;
|
```

```
% _ parentof _ : a little complicated because we need a special case
%       for ISA_COMPONENTs to prevent endless recursion of isa's
%
(forall p,c:Contents; o:OFilter | p /= ISA @
      (p->o) parentof c <=>
   (exists a:ISA_COMPONENT |
         a in Contents /\ GetFilter(a) accepts o @
         GetTerminal(a,2) = p /\ GetTerminal(a,1) = c))
/\
(forall p,c:Contents; o:OFilter | p = ISA @
      (p->o) parentof c <=> (c.Role in ran isa_arc /\ p /= c /\
     not (exists a:ISA_COMPONENT |
         a in Contents /\ GetFilter(a) accepts o @
         GetTerminal(a,2) = p /\ GetTerminal(a,1) = c)));

% paths_btwn() : enumerates all the paths between a child and an ancestor
forall c,p:Contents; o:OFilter @ paths_btwn(c,p,o) = {s:PATH |
     first(first(s(1))) = c /\
     (forall i:1..#s @
         s(i) in ( _ parentof _ ) /\
         second(s(i)) = (if i=#s then p else first(first(s(i+1)))) /\
         second(first(s(i))) = (if i=1 then o else
          Intersection {x:OFilter | (exists j:1..i-1 @
                x = second(first(s(j))))}))};

% _ ancestorof _ : recursively accents the _parentof_ relation while
%         accumulating a more restrictive filter
%
forall c,p:Contents; o:OFilter @
      (p->o) ancestorof c <=> paths_btwn(c,p,o) /= {};

% _ isa _ : simply the disjunct of the _ancestorof_ and
%           identity relationships
%
forall p,c:Contents; o:OFilter @ (c->o) isa p <=>
                                  ((p = c) \/ ((p->o) ancestorof c));

% details of the level 1 components
TOP.Name = T;
TOP.Level = 1;

NODE.Name = Node;
NODE.Level = 1;
NODE.Role in ran node;

ARC.Name = Arc;
ARC.Level = 1;
#(arc~(ARC.Role)).Terminals = 0;

ISA.Name = IsA;
ISA.Level = 1;
GetTerminal(ISA,1) = TOP;
GetTerminal(ISA,2) = TOP;

CONTEXT.Name = Context;
CONTEXT.Level = 1;
members~((node~(CONTEXT.Role)).Contents) = {}; %CONTEXT has no members

% the type hierarchy for the level 1 components
TOP->UniversalOntology parentof NODE;
TOP->UniversalOntology parentof ARC;
```

```
ARC->UniversalOntology parentof ISA;
NODE->UniversalOntology parentof CONTEXT;

% The only level 1 objects are TOP, NODE, ARC, ISA, and CONTEXT and their
% isa arcs that define the type hierarchy
#{c:COMPONENT | c.Level = 1} = 9;

% there are no cycles over the isa relationship
forall c:Contents @ not (c->UniversalOntology ancestorof c);

% everything is a NODE or ARC, but only one of these
forall c:Contents @
  (c->UniversalOntology isa NODE \/ c->UniversalOntology isa ARC) /\
  (c->UniversalOntology isa NODE) => not (c->UniversalOntology isa ARC)/\
  (c->UniversalOntology isa ARC ) => not (c->UniversalOntology isa NODE);

% CONTEXTs contain only other objects in the graph
forall x:CONTEXT_COMPONENT | x in Contents @
  forall m:COMPONENT | m incontext x @ m in Contents;
------------------------------------------------------------------------
```

5.9 VALIDATORS are Constraints

Requirement 15 necessitates a mechanism to constrain components. For example, arcs need to be constrained so that they can't terminate on themselves. Validators act as constraint rules attached to components. A Validator is merely a boolean function applied to a Component in the context of a graph. Since these constraints will apply to all subtypes (as defined by ISA_COMPONENT, below), it is necessary to use two Components as arguments – the first is the exact object under inspection, and the second is the supertype that actually contains the validator. In general, in the implementation, it is expected that actual validators would be function objects (Laufer 1995; Nelson 1995).

```
VRES ::= PASS | FAIL

VALIDATOR == COMPONENT & COMPONENT & GRAPH0 & OFilter --> VRES
```

The remainder of this section lists a few specific VALIDATORs that will be needed shortly. Their definitions follow.

No arc may refer to itself. This rule will be added to the definition of an ARC.

```
NoSelfReference: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  NoSelfReference(apply,owner,g,o) =
    if (forall t:second(| (arc~(apply.Role)).Terminals |) @
                                 apply /= t.Anchor)
         then PASS
         else FAIL
```

The following VALIDATOR template restricts the arity of an arc. It is actually a function that takes a positive integer argument and returns a VALIDATOR that restricts the arity to be equal to the value of the argument. This can be easily implemented as a C++ function object (with state – the argument). Similar functions could easily be defined for *greater-than*, *less-than*, etc. Implementations should contain these in a validator library.

```
ArityEquals: N1 --> VALIDATOR
---------------
forall n: N1 @
  ArityEquals(n) = (mu v:VALIDATOR |
    forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
      apply in g.Contents /\ owner in g.Contents @
      v(apply,owner,g,o) =
   if (#((arc~(apply.Role)).Terminals) = n)
         then PASS
         else FAIL)
```

The Directed VALIDATOR specifies a binary from-to arc. This will be used to restrict the *isa* arc (**Requirement 8**).

```
Directed: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  Directed(apply,owner,g,o) =
    if (ArityEquals(2)(apply,owner,g,o)=PASS  /\
   GetTerminalDir(apply,1) = FROM  /\
   GetTerminalDir(apply,2) = TO)
         then PASS
         else FAIL
```

The Upward VALIDATOR is a specialization of the Directed VALIDATOR that must be *from* an object at a level lower or equal to the level of the *to* side. Also, the level of the *isa* arc itself must be the same as that of the *from* side.

```
Upward: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  Upward(apply,owner,g,o) =
    if (Directed(apply,owner,g,o)=PASS          /\
      (GetTerminal(apply,1)).Level >=  (GetTerminal(apply,2)).Level /\
      apply.Level <= (GetTerminal(apply,1)).Level)
        then PASS
        else FAIL
```

An arc cannot refer to anything outside the graph it is contained in. This validator is related to the NoDanglingTerminals validator but allows dangling terminals and explicity checks for graph membership. (NoDanglingTerminals does not.)

```
ConfinedToGraph: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  ConfinedToGraph(apply,owner,g,o) =
    if (forall i:N | i<=#(arc~(apply.Role)).Terminals @
        GetTerminal(apply,i) in (g.Contents || {NULLCOMPONENT}))
        then PASS
        else FAIL
```

A component cannot override a CONSTANT attribute (**Requirement 22**).

```
NoConstAttrOverrides: VALIDATOR
--------------
forall apply, owner: COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  NoConstAttrOverrides(apply,owner,g,o) =
    if (forall a:apply.Attributes @
      not (exists c2:g.Contents | (c2->o)->apply in g.ancestorof0 @
   forall a2:c2.Attributes | a2.Name = a.Name @ CONSTANT in a2.Flags))
        then PASS
        else FAIL
```

Arcs must conform to their parents' arcs; that is, the child's arity must be greater than or equal to the arity of the parent, and all terminating objects in the child must be sub-types (or equal) of the corresponding (and non-missing) types of the parent's terminating objects. This restriction is a corollary of **Requirement 14**.

```
ArcConformance: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  ArcConformance(apply,owner,g,o) =
    if (forall c2:ARC_COMPONENT | c2 in g.Contents /\
                                  (c2->o)->apply in g.ancestorof0 @
    % subtypes must have equal or more terminals
    #(arc~(c2.Role)).Terminals <= #(arc~(apply.Role)).Terminals /\
    % subtypes must type-conform at each terminal
    (forall i:Z | i<=#(arc~(c2.Role)).Terminals @
      (GetTerminal(apply,i)->o)->GetTerminal(c2,i) in g.isa0))
        then PASS
        else FAIL
```

Contexts are strictly nested and no component may be a member of more than one context (**Requirement 12**). Note that the previous statement is a bit redundant: since contexts are themselves components, it is sufficient to say no component may be a member of more than one context. This rule will be added to the definition of a CONTEXT.

```
NestedContexts: VALIDATOR
--------------
forall apply, owner: CONTEXT_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  NestedContexts(apply,owner,g,o) =
    if (forall c:g.Contents @
        % no component is a member of more than one context
    c in members~((node~(apply.Role)).Contents) =>
      (forall c2:CONTEXT_COMPONENT | c2 in g.Contents /\ c2 /= apply @
        not(c in members~((node~(c2.Role)).Contents))))
         then PASS
         else FAIL
```

All *isa* arcs must have a filter that is a subset of each of its supertype *isa* arcs. The restriction is necessary to support **Requirement 18**, overlayed type lattices.

```
IsaSubtypesHaveSmallerFilters: VALIDATOR
--------------
forall apply, owner: ISA_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
    IsaSubtypesHaveSmallerFilters(apply,owner,g,o) =
      if (forall p:ISA_COMPONENT |
  p in g.Contents /\ (p->UniversalOntology)->apply in g.ancestorof0 @
  GetFilter(apply) subseteq GetFilter(p))
        then PASS
        else FAIL
```

An arc with the NoDanglingTerminals contraint must have all its terminals linked to a valid object in the graph (none can be uninstantiated or "dangling"). Natureally, this constraint applies to all *isa* arcs. Note that this is related to the ConfinedToGraph validator: ConfinedToGraph allows dangling terminals while this constraint does not. Also, NoDanglingTerminals does not explicitly check for membership in the graph.

```
NoDanglingTerminals: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
    NoDanglingTerminals(apply,owner,g,o) =
      if (forall i:N | i<=#(arc~(apply.Role)).Terminals @
     GetTerminal(apply,i) in g.Contents)
         then PASS
         else FAIL
```

The following validator is not required by the specification, but it serves as an example of how to restrict graphs in ways that are commonly required by various graphical formalisms.

NoArcBtwnArcs disallows arcs between arcs (except ARC itself and all ISA arcs). This validator can be used at the ARC level to force first order graphs.

```
NoArcBtwnArcs: VALIDATOR
--------------
forall apply, owner: ARC_COMPONENT; g: GRAPH0; o:OFilter |
  apply in g.Contents /\ owner in g.Contents @
  NoArcBtwnArcs(apply,owner,g,o) =
    if (((apply->o)->g.ARC in g.isa0 /\ apply /= g.ARC /\
               not((apply->o)->g.ISA in g.isa0)) => (forall t:TERMINAL |
     t in ran (arc~(apply.Role)).Terminals @
                                  (t.Anchor->o)->g.NODE in g.isa0))
         then PASS
         else FAIL
```

5.10 Finally, a Typed Graph

Now that validators have been specified and several instances of validators have been defined, a TYPED_GRAPH can be described. A typed graph has several constraints over GRAPH0:

- The reason for CONSTANT flag on attributes can now be realized: a locked attribute/value pair is simply one that cannot be overridden (**Requirement 22**).

- No arc or *isa* arc may have a terminal pointing at itself.
- No arc or *isa* arc may point outside the graph.
- All arcs' and *isa* arcs' terminal objects types must conform to its parent's corresponding terminal: each terminal type must be a subtype of its parent's corresponding terminal type (**Requirement 14**).
- All *isa* arcs must point upward (this implies they are also binary and directed); that is, level of the object at the *from* terminal must be less than or equal to the level of the object at the *to* terminal.

All of the above rules are applied as constraints (VALIDATORS) on the four basic components in the graph. Furthermore, all these validators must be satified for every subtype of the four basic components; and since everything is a subtype of one or two of these, the entire graph is appropriately constrained.

```
---TYPED_GRAPH----------------------------------------------------------
| GRAPH0;
|-------------
| % T restrictions
| {NoConstAttrOverrides} subseteq TOP.Constraints;
|
| % arc restrictions
| {NoSelfReference, ConfinedToGraph, ArcConformance}
|       subseteq ARC.Constraints;
|
| % an isa arc is binary upward-directed arc (recall that ISA isa ARC, and
| % "inherits" its constraints)
| {Upward, NoDanglingTerminals} subseteq ISA.Constraints;
|
| % context restrictions
| {NestedContexts}
|       subseteq CONTEXT.Constraints;
|
| % all components satisfy their constraints
| exists g:GRAPH0 | g.Contents = Contents @
|   forall c,p:g.Contents | (c->UniversalOntology) isa p @
|     (forall s:paths_btwn(c,p,UniversalOntology) @
|       forall v:VALIDATOR | v in p.Constraints @
|    v(c,p,g,second(first(s(#s)))) = PASS)
|     /\
|     (forall v:VALIDATOR | v in c.Constraints @
|       v(c,c,g,UniversalOntology) = PASS)
|
-----------------------------------------------------------------------
```

The following is a consistency corollary of the above TYPED_GRAPH schema and can easily be proved from it. Every component is the same top-level type as all of its supertypes.

```
forall g:TYPED_GRAPH @ forall c1,c2:g.Contents |
                              (c1->UniversalOntology)->c2 in g.isa0 @
   (c1->UniversalOntology)->g.NODE in g.isa0 <=>
                        (c2->UniversalOntology)->g.NODE in g.isa0 /\
   (c1->UniversalOntology)->g.ARC  in g.isa0 <=>
                        (c2->UniversalOntology)->g.ARC  in g.isa0
```

5.11 The Operators

This section lists operators on graphs.

5.11.1 Attribute Operators

The following functions determine the value of a particular attribute given a TYPED_GRAPH, a component, and an attribute name. The first function, getAttr searches for an attribute – looking first in the actual component, then attempts to select the highest priority attribute that is not hidden by an intervening attribute. Note that locked attributes are already taken care of in the TYPED_GRAPH schema. Note also that this function is not necessarily deterministic: this problem will have to be addressed in the future.

```
GETATTR_RESULT ::= Attr<> | NULLr
```

```
| getAttr: TYPED_GRAPH & COMPONENT & ATTRIBUTE_NAME --> GETATTR_RESULT
|--------------
| forall g:TYPED_GRAPH; c:COMPONENT; a:ATTRIBUTE_NAME | c in g.Contents @
|   getAttr(g,c,a) =
|     if a in {a1:ATTRIBUTE_NAME | exists at:c.Attributes @ a1 = at.Name}
|       % c actually has the attribute
|       then (mu x:c.Attributes | (a = x.Name))
|       else if {x:GETATTR_RESULT |
|         forall c2:g.Contents |
|                             (c2->UniversalOntology)->c in g.parentof0 @
|                                    x=getAttr(g,c2,a)} = {}
|   % attribute not found
|   then NULLr
|   % attribute found in one of the ancestors:
|   else (mu x:ATTRIBUTE | forall c2:g.Contents |
|         (c2->UniversalOntology)->c in g.parentof0 /\
|         (exists y:ATTRIBUTE @ y=Attr~(getAttr(g,c2,a))) @
|          exists y:ATTRIBUTE | y=Attr~(getAttr(g,c2,a)) @
|            forall c3:g.Contents |
|              (c3->UniversalOntology)->c in g.parentof0 /\
|              (exists z:ATTRIBUTE @ z=Attr~(getAttr(g,c3,a))) /\
|              (not ((c3->UniversalOntology)->c2 in g.ancestorof0)) /\
|              (not ((c2->UniversalOntology)->c3 in g.ancestorof0)) @
|                exists z:ATTRIBUTE | z=Attr~(getAttr(g,c3,a)) @
|                 y.Priority<=z.Priority /\ x=y)
```

The getValue function is similar to the above function, but simply returns a value if one can be found.

```
GETVALUE_RESULT ::= Val<> | NULLv
```

```
| getValue : TYPED_GRAPH & COMPONENT & ATTRIBUTE_NAME --> GETVALUE_RESULT
|---------------
| forall g:TYPED_GRAPH; c:COMPONENT; a:ATTRIBUTE_NAME | c in g.Contents @
|       getValue(g,c,a) = if getAttr(g,c,a)=NULLr
|                      then NULLv
|                      else (Attr~(getAttr(g,c,a))).Value
```

setAttr takes a component and an attribute and merely adds the attribute to the component's set of attributes, replacing any attributes whose name might have matched.

```
---setAttr-----------------------------------------------------------
| Delta TYPED_GRAPH;
| c?: COMPONENT;
| attr?: ATTRIBUTE
|---------------
| c? in Contents;
| Contents' = (Contents \ {c?}) || {(mu c2:Contents |
|       c2.Name  = c?.Name      /\
|       c2.Level = c?.Level     /\
|       c2.Role  = c?.Role      /\
|       c2.Attributes = (c?.Attributes \
|          {a:c?.Attributes | a.Name = attr?.Name}) ||
|          {attr?})}
---------------------------------------------------------------------
```

5.11.2 Graph Modifying Operators

Its a simple procedure to add a node to a graph:

```
---addNode-----------------------------------------------------------
| Delta TYPED_GRAPH;
| node?: NODE_COMPONENT;
| type?: NODE_COMPONENT;
| o?: OFilter;
|---------------
| TYPED_GRAPH';
| type? in Contents;
| not (node? in Level1Objects);
| Contents' = Contents || {node?, (mu a:ISA_COMPONENT |
|   GetTerminal(a,1)=node? /\ GetTerminal(a,2)=type? /\ GetFilter(a)=o?)};
---------------------------------------------------------------------
```

Note that, due to the invariants on the graph, the node cannot have any non-ISA arcs in the graph that refer to it immediately after its insertion. Note also that, by virtue of the definition of NODE_COMPONENT, the new node is a subtype of NODE (and therefore there exists an chain of ISA links from the new node to NODE.

It is just as simple to add a new arc:

```
---addArc-----------------------------------------------------------------
| Delta TYPED_GRAPH;
| arc?: ARC_COMPONENT;
| type?: ARC_COMPONENT;
| o?: OFilter;
|---------------
| TYPED_GRAPH';
| type? in Contents;
| not((arc?->UniversalOntology) isa ISA);
| not(arc? in Level1Objects);
| Contents' = Contents || {arc?, (mu a:ISA_COMPONENT |
|         GetTerminal(a,1)=arc? /\
|         GetTerminal(a,2)=type? /\
|         GetFilter(a)=o?)};
--------------------------------------------------------------------------
```

But note that there is more going on here than meets the eye. Because of the constraints on a typed graph, the arc must refer only to nodes that are already members of the graph contents.

It must also be possible to change the type of node or arc. However, since this system uses multiple inheritance, "changing" the type makes little sense. Instead, one can set an object to be some type (by adding an *isa*-arc). Removing a type from a component's set of parent types is just a matter of deleting a particular *isa* arc and can be accomplished by using *remove*, as shown below.

```
---setComponentType-------------------------------------------------------
| Delta TYPED_GRAPH;
| c?: COMPONENT;
| newType?: COMPONENT
|---------------
| TYPED_GRAPH';
| ((c?->UniversalOntology) isa NODE /\
|                            (newType?->UniversalOntology) isa NODE) \/
| ((c?->UniversalOntology) isa ARC  /\
|                            (newType?->UniversalOntology) isa ARC );
| c? in (Contents \ Level1Objects);
| newType? in Contents;
| Contents' = Contents ||
|   {(mu a:ISA_COMPONENT | GetTerminal(a,1)=c? /\
|                                          GetTerminal(a,2)=newType?)};
--------------------------------------------------------------------------
```

Removing any component from a graph entails removing all references to it from arcs (including *isa* arcs) and contexts. One problem is that if the removed node was a supertype of some existing component, then the subtype component will be left with a dangling *isa* chain. This is prevented by disallowing it in precondition.

```
---remove---------------------------------------------------------------------
| Delta TYPED_GRAPH;
| c?: COMPONENT;
|---------------
| TYPED_GRAPH';
| c? in (Contents \ Level1Objects);
| not (exists c2:Contents @ (c2->UniversalOntology) isa c?);
| % the target is removed
| Contents' = Contents \ {c?};
| % either arcs are dangling or removed
| forall a,oa:ARC_COMPONENT |
|   oa in Contents /\ a in Contents' /\ a.Name=oa.Name @
|     forall i:N @ GetTerminal(oa,i)=c? => GetTerminal(a,i)=NULLCOMPONENT;
| % the removed component is deleted from any enclosing context
| forall cxt,ocxt:CONTEXT_COMPONENT |
|   c? incontext ocxt /\ ocxt.Name=cxt.Name @
|     not (c? incontext cxt);
------------------------------------------------------------------------------
```

5.12 First-order Typed Graphs

It is not particularly common in visual graph languages to allow arcs between arcs, and this restriction may easily be applied to this typed graph theory. Such a graph is called a FIRST_ORDER_TYPED_GRAPH, and is just a constraint on an ordinary TYPED_GRAPH: all arc terminals must refer to nodes only. Note that this does not constrain the type system, as *isa* arcs can refer to nodes or links (or other *isa*s).

```
%% state-schema FIRST_ORDER_TYPED_GRAPH

FIRST_ORDER_TYPED_GRAPH =^= [TYPED_GRAPH |
    {NoArcBtwnArcs} subset ARC.Constraints
    ]
```

This schema demonstrates just how simple it can be to constrain a graph to a particular syntax.

5.13 Summary

This chapter has described the specification of the Constraint Graphs system. The Constraint Graphs system consists of a TypedGraph which acts as container of graph COMPONENTs. A Typed Graph also has the responsibility to maintain various invariants among its components. Every graph component has a name, a level, a set of attributes and a set of constraints. There are four types of components: nodes, contexts, arcs, and *isa* arcs. Attributes are named values where the type of the value is left open; however, while the value of an attribute can be overridden when it is inherited, its type cannot be changed.

The specification given in this chapter is a clear and unambiguous description of the theory and the framework and supports **Objective 1** and **Objective 2**. This is an important result of the specification, but equally important, the specification was used during the development of the theory to work with the various possible variants before committing to an implementation. The cost of backtracking to design during the implementation stage is often prohibitively expensive (so it not always done when it should be, compromising the software design). Working with the specification, and ironing out as many problems as possible in the specification minimizes the commitment to a particular design, encourages backtracking to design, and therefore can lead the a better final design. During development, several inconsistencies and possible generalizations were detected at the specification stage; the minimal commitment at this stage made it easy to backtrack to the preliminary design to correct the errors. For example, early versions of the theory had a much less clean fundamental type lattice (*isa* was not originally a subtype of an ordinary arc), but the specification made the similarities and the (now-obvious) subtype relationship clear, and this observation was propagated back into an improved theory. Many iterations of the specification lead to a much stronger theory and final design of the framework. The specification will continue to be used as pre-implementation testing ground for framework design and theory changes.

The drawback of the specification is the time and effort it takes to produce a specification. The process of creating a specification can involve nearly the effort it takes to produce a prototype-level implementation. This is particularly true when the specification language is not a perfect match for the domain (as happened with the Z language and the convoluted way in which one has to specify inheritance relationships – see the introduction to this chapter). However, for a research area in which the direction is not clear, the clarity and unambiguous nature of the specification seems well worth the effort (as it was in this case).This chapter does *not* describe the interface or any kind of user interaction. That is a separate layer, and is discussed in the next chapter.

Chapter 6

Design and Implementation

The previous chapter presented a formal specification of the theory behind the Constraint Graphs system. This chapter describes the design and implementation of one particular program that fulfills that specification. This chapter also extends the description to include a user interface, which is a separate layer within the system as per **Requirement 1** (see Figure 16 in Section 6.1). Due to space limitations and its subservient focus, the user interface design is only superficially described in Section 6.2 (and is not formally specified at all); but an earlier version of the user interface is documented elsewhere (Kremer 1993). The actual C++ implementation from the specification in chapter 5 is described in Section 6.3. The implementation is a collection of related C++ classes, each corresponding to a component of the specification. While the description of the class libraries gives the reader a very good idea of the software design and structure, it does not shed much light on how the tool is used in a practical sense. Section 6.4 describes how the software can be used to implement a visual language using a type lattice, arc termination constraints, and ad-hoc constraints, as well as how to use the ontological levels to improve the user interface. Finally, Section 6.5 is a catch-all which describes some of the interesting technical details in the implementation.

6.1 Independence of User Interface and Constraint Graph

In the overall structure of the design, there is an abstract Constraint Graph (described in chapter 5) which implements all of the requirements of Section 4.3. While this abstract graph presents a very specific software interface, it implements no user interface at all.

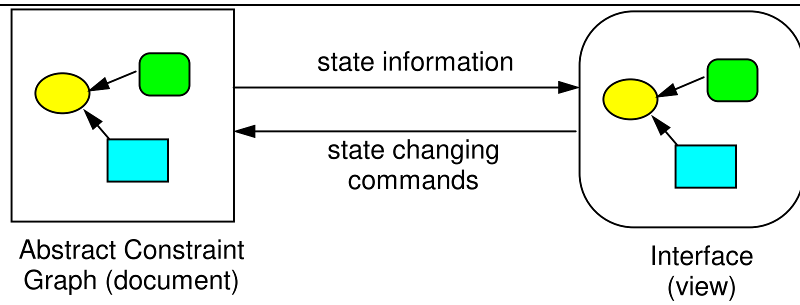


Figure 16: Separate abstract graphs and interfaces communicate state information and state-changes commands.

There is a separate and independent user interface (see Figure 16) which is solely responsible for

1. drawing a version of the abstract graph on the screen,
2. collecting input from the user,
3. translating the input into operations on the abstract graph, and
4. updating the on-screen version of the graph to reflect any state changes resulting from graph operations.

The design closely tracks the Model-View-Controller pattern described in (Buschmann et al. 1996) and used in the Smalltalk class library; however, the View and Controller are bundled into a single role (often called the Document-View variant), which is not uncommon (Gamma et al. 1995). This pattern has several advantages, for example:

- Many different interface packages may be layered on top of the abstract constraint graph.
- Several different sub-views (or "filters") of the abstract constraint graph may be implemented on a single abstract graph (see Figure 17).

The constraint graph is responsible for all information about relationships between *components* (the collective name for nodes and arcs), including the type lattices. It is also responsible for maintaining arbitrary information about all components in the form of attribute/value pairs associated with each component. Attribute/value pairs are inherited though the type lattice. The constraint graph is *not* responsible for maintaining physical information about components such as physical size and location; it is only responsible for relationships. However, a view may store some physical information (color, for example) in the constraint graph's components' attribute/value pairs list if this information is intended to be global between views and is intended to be inherited by subtypes. Size and location are attributes that should be neither global between views nor inherited by subtypes: The same component in more than one view may not necessarily occupy the same coordinates. Furthermore, inheritance of location may lead to the absurd situation where every object occupies the same default location, since every component is a subtype of T (top).

Views, or interfaces, are responsible for physical information such as location and size of node. As already mentioned, views may store some physical information in the constraint

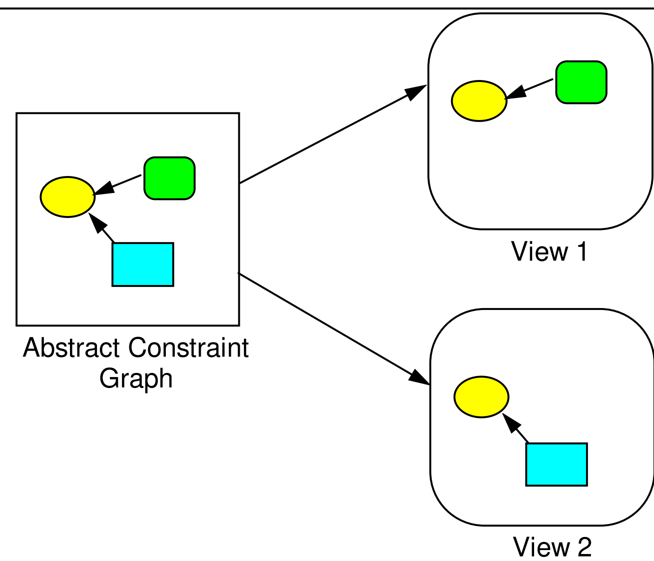


Figure 17: A single abstract graph with multiple sub-views in the interface.

graph's components' attribute/value pairs, provided the information is intended to be view-global and inherited through the type lattices.

The independence of the abstract constraint graph and interface is reflected in the storage format as described in Section 6.5.1.

6.2 The CMap Class Library

The previous section described the independence of the graphical user interface and the abstract constraint graph. This section fills in details of the structure of the independent interface layer and how this independence in achieved.

An independent graphical class library was built to support drawing concept maps on the screen. The class library is called CMap (for Concept Map), and was tested by building a simple stand-alone concept map drawing program called KSIMapper. KSIMapper is only useful as a drawing tool and does not support constraints of any kind. It does no more than provide a basic windowing and interface event environment in which the CMap class library objects can operate.

A slightly simplified version of the CMap class library is shown in Figure 18. Graphical objects are divided into visual graphics and behavioural graphics. All behavioural graphics implement interface behaviours and are never directly drawn to the screen (they are *controllers* in the model-view-controller pattern). Behavioural graphics all *delegate* drawing to an object which is a subtype of visual graphic (the directed arc labeled "refers-to" in Figure 18). Since all programmatic references are always to behavioural graphics, this scheme allows graphical objects to alter the specific drawing class without destroying the integrity of any enclosing structure. For example, a Node may use an *EllipseShape* or the *RectangleShape* (both subclasses of *Shape*) as its visual delegate, but a user can easily swap between the two because no object (other than the Node) refers directly to the delegate.

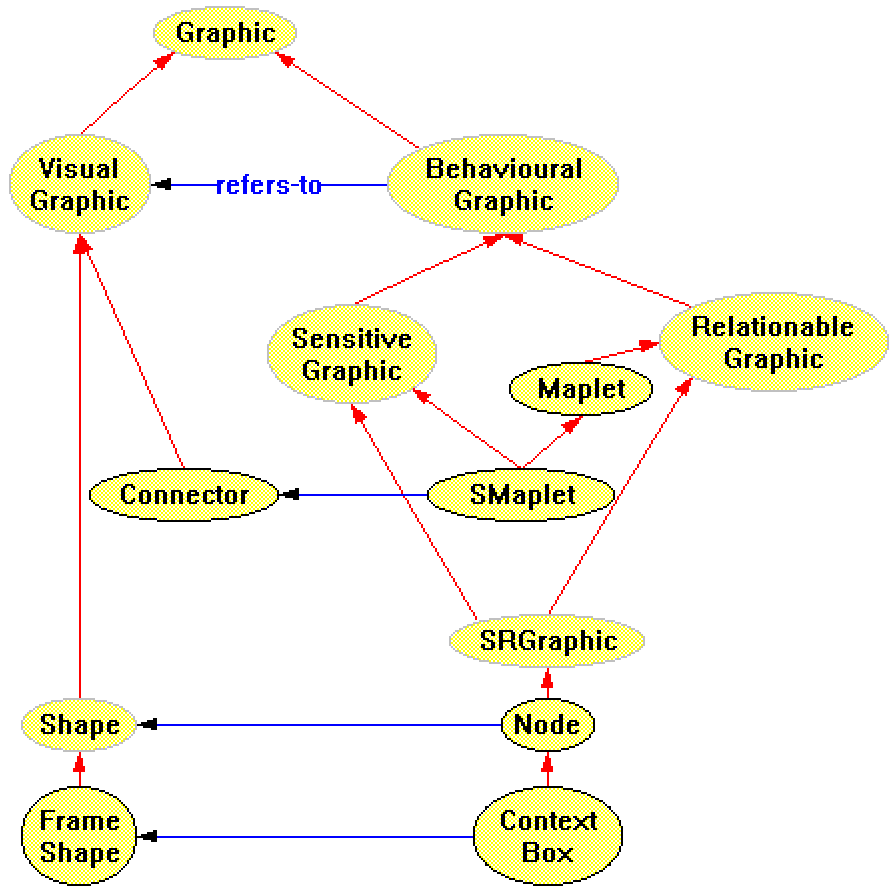


Figure 18: A simplified version of the CMap class library. Abstract classes are pictured with grayed borders; concrete classes are pictured with solid black borders.

The horizontal directed arc labeled "refers-to" in Figure 18 represents the actual delegate pointer between behavioural graphics objects and their delegate visual graphics. However, specific concrete behavioural graphics demand references to specific subtypes of VisaulGraphic. The unlabeled directed horizontal arcs in Figure 18 represent these "virtual" references between specific behavioural graphics and specific visual graphics. The behavioural graphics all use the more general delegate pointer in BehaviouralGraphic, but always access the visual graphic through a local method which

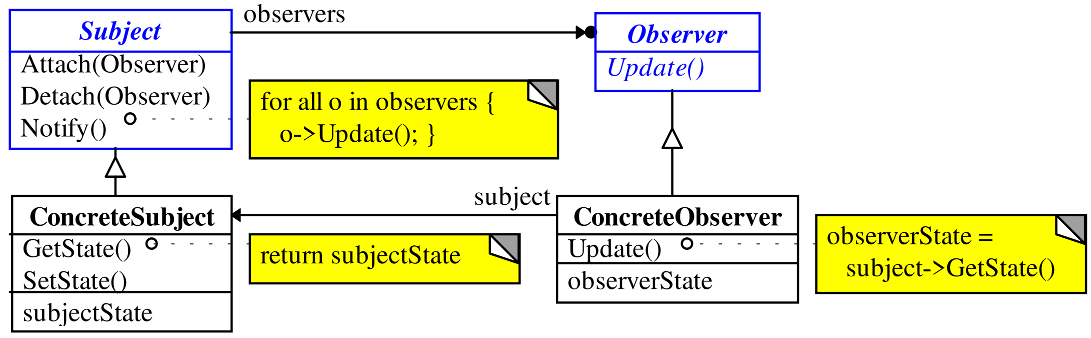


Figure 19: The observer pattern (Gamma et al. 1995)

ensures the type safety of the reference. For example, Node never uses the delegate pointer BehaviouralGraphic::Visual, but instead uses its local method,

```
Shape* Node::getShape()
```

which does the necessary run-time type checks and dynamic type caste to ensure the type safety of any operations.

There are specific behavioural classes which map onto the base types in Constraint Graphs: CMap's Nodes, ContextBoxes, and SMaplets (Figure 18) map onto Constraint Graphs' Nodes, Contexts, and Arcs (Figure 11 on page 64). There is no need for a special CMap class to model Constraint Graphs' *Isa* arcs: plain SMaplets will do.

Every component of the CMap graph is an observer of the corresponding component in the Constraint Graphs graph (Figure 20) according to the Observer pattern (Gamma et al. 1995) shown in Figure 19. Whenever a component in the Constraint Graph changes in any way, it calls its Notify() method, which sends the Update() message to each of the CMap components which are observing it. In this way, the visual image of every Constraint Graphs component is always easily and efficiently updated to reflect state changes, even if there are multiple views (CMap graphs) or multiple occurrences of the image of a Constraint Graphs component on a single view.

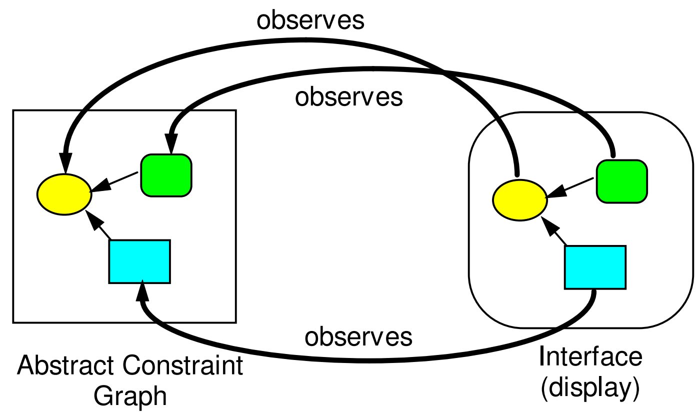
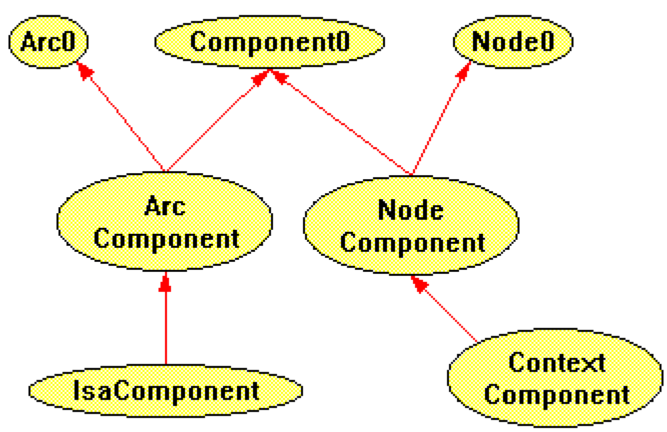


Figure 20: Interface elements are observers of corresponding components of the abstract constraint graph. Although only observations of nodes are pictured here, arcs also observe the corresponding arcs in the constraint graph.

6.3 The Constraint Graphs Class Hierarchy

This section describes the class hierarchy used in the Constraint Graphs implementation based on the specification in chapter 5. Figure 21 is a diagram of the class hierarchy.

The central part of Figure 21 is the sub-graph consisting of the *isa* tree rooted at the Component0 class:



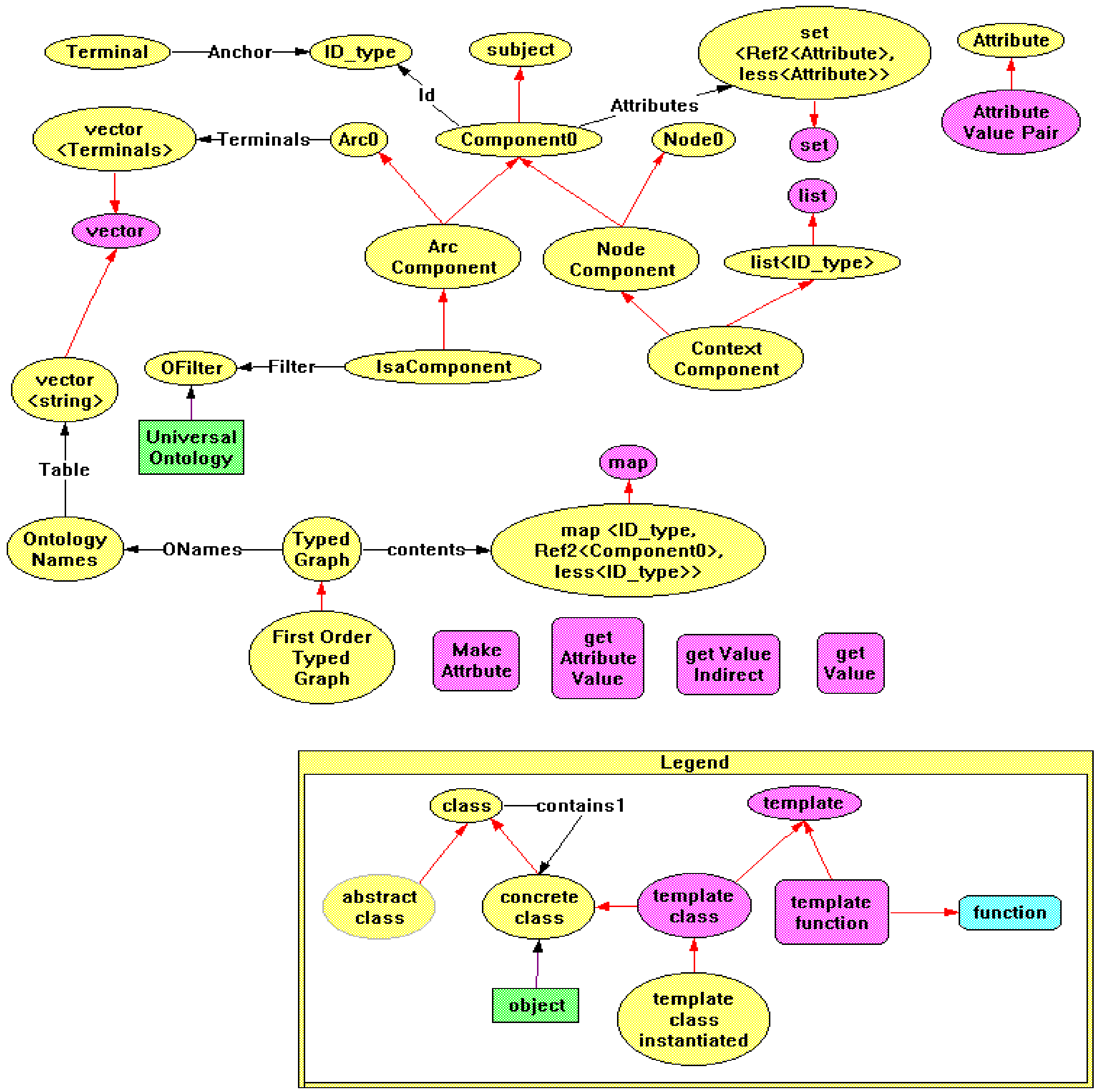
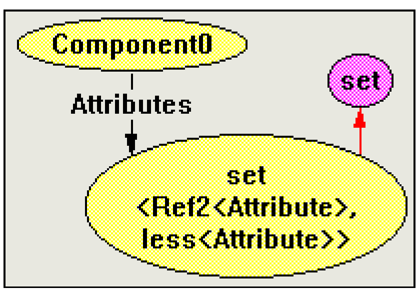


Figure 21: The class hierarchy for the ConstraintGraphs implementation

which is a close match with Figure 11 on page 64 (the base type lattice) and also very closely tracks the Z schemata in Sections 5.4 through 5.6. Technically, there is no reason why Arc0 and Node0 have to exist: the two classes should normally be removed and their functionality moved to ArcComponent and NodeComponent respectively. However, the constraints of the Z specification language require their existence (see the introduction to chapter 5), and the design merely tracks the specification's structure in this case. Having the class structure of the implementation model the structure of the specification also helps the hypermedia documentation (Section 6.5.4) in that one can hyperlink between corresponding structures in the specification and implementation.

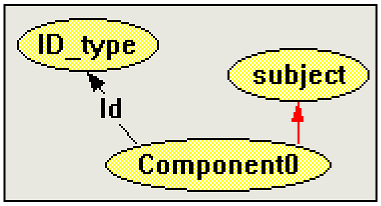
6.3.1 Component0

Component0 forms the base class for ArcComponent and NodeComponent and maps to Component in the specification (Section 5.6). As such, Component0 must have a name, a level, a set of attributes, and a set of constraints (or validators). Both name and level are left out of Figure 21 to avoid clutter. The constraints are also conspicuously absent: the implementation differs from the specification in that the implementation considers a constraint to be just a particular type of attribute/value pair. In other words, instead of having a specific constraint list, this implementation merely looks up the attribute "constraints" to find the constraint list.



Component0's attribute/value pairs are implemented as an STL set containing references to elements of the Attribute class, which is merely an attribute label (modeling attributes without values). The Attribute class is itself subclassed by a template which extends the base attribute to include values which are strictly-typed, according to **Requirement 21**. Variably-typed attributes are very difficult to express in Z, so this detail is glossed over in the specification.

An obvious alternative design is to implement attribute/value pairs as an STL map from a class string to a value class; but this design does not easily address the problem of type restriction on the value parts without breaking encapsulation.

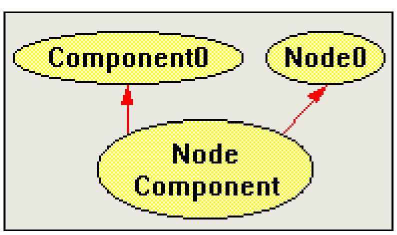


In addition, Figure 21 shows Component0 as having two relationships which do not appear in the specification. First, Component0 is a subtype of the subject class, which makes it's objects "observable" by other observer-class objects in the interface according to the observer pattern described in Section 6.1. Secondly, Component0 has a member, "Id", which is a unique identifier used to identify the object as a specific individual. In the Z specification, the identity function is implicit. Implementations often use the memory address as identity; but this implementation makes no such assumptions about address space because

- objects may be copied and still retain their identity (common in STL containers), and
- objects may migrate among address spaces and among machines (see Section 8.3).

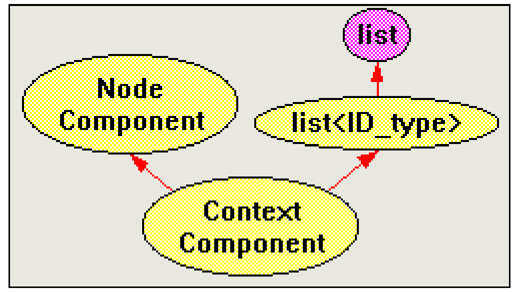
Thus, a unique identifier is required for every component. The identifiers are only unique within a single Constraint Graph, and the TypedGraph class is responsible for generating them.

6.3.2 Nodes

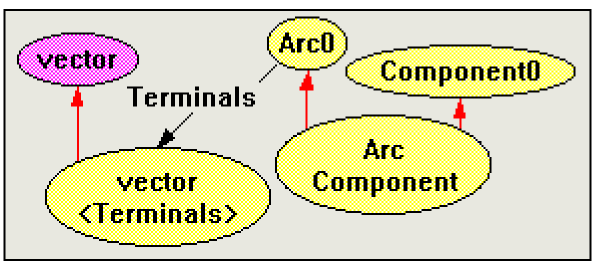


The NodeComponent class itself does not do a lot. It is not much more than a placeholder in the type lattice for a specialization of a Component0 that is mutually exclusive from an ArcComponent. NodeComponent can be subclassed to provide specialized behaviour. A case in point is ContextComponent.

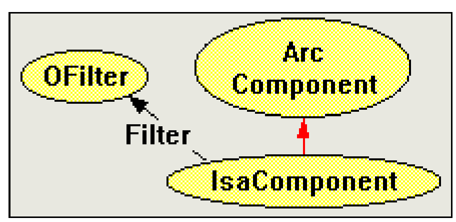
ContextComponent is a subclass of NodeComponent which extends it by adding the ability to contain sub-graphs (collections of other components). This is done by specializing an STL list of ID_types, which constitutes a list of references to other components. The list is restricted to contain only valid identifiers of components of the same graph which the ContextComponent object is a part of, as per specification Section 5.8. In addition, the list must not contain any unique identifiers that are contained in any other ContextComponent objects, which forces the ContextComponents to be strictly nested as per specification Section 5.10.



6.3.3 Arcs



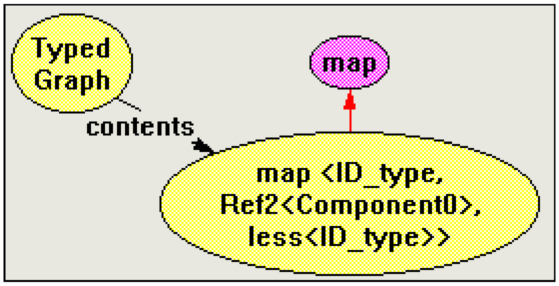
The ArcComponent class is another subtype of Component0, and extends it by adding an STL vector of Terminals. The Terminal class is simply an association of a component reference (a unique identifier) and a direction as required by the specification in Section 5.4.



The IsaComponent class extends ArcComponent by adding an ontology filter as per the specification in Section 5.5 and **Requirement 18**. The ontology filters are implemented as a long integer and are treated as a bit vector. This treatment is detailed in Section 6.3.4.

IsaComponents are also restricted to be binary and directed (Section 5.5 and **Requirement 8**).

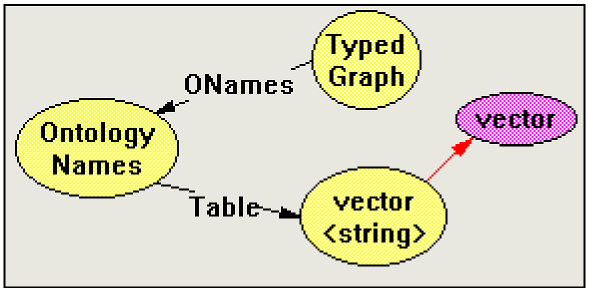
6.3.4 Typed Graph



The TypedGraph class is the implementation of the TypedGraph specification in Section 5.10. In fact, it is the merge of both the TypedGraph specification and the Graph0 specification in Section 5.8: Declaration dependencies could not be accommodated in the specification as easily as in the actual implementation. The TypedGraph class is primarily responsible to be a container for all the graph components. It does this using a STL map member that maps between ID_type (unique component identifiers) and indirect references to the corresponding components.

Since the TypedGraph class is frequently used to look up components by their unique identifiers, the choice of data structure here is critical. An STL map is a good choice because it is a fast version of a balanced binary tree – a red-black tree (Nelson 1995). Binary tree lookups and insertions have $O(\log_2 n)$ complexity. This is worse than a vector's constant complexity, but a vector is not appropriate because unique identifiers (the logical index) cannot be re-allocated after a component is deleted (since this may confuse external references). A graph implemented with a vector may eventually end up with much more empty space than actual data. At the other end of the spectrum is a list, but its linear lookup complexity is not efficient enough for the frequent lookups required. An STL hash table might be a better choice than a map, but hash tables are not yet part of the STL (Barreiro, Fraley & Musser 1995).

The TypedGraph class is also responsible for maintaining the invariants given in the specification in Sections 5.8 and 5.10. Also, as per Section 5.8, all typed graphs always contain the Constraint Graph base types (Node, Arc, Context, and Isa), as well as the *isa* arcs that constitute their type lattice (Figure 11 on page 64).



The specification deals with the ontology tags on *isa* arcs in a very abstract manner. The implementation has no such luxury. In the implementation, the IsaComponent's ontology tag set is implemented as a simple long integer which is treated as a bit vector. This is definitely not appropriate for the user, for the user will want ontology names to be strings instead of bit offsets. But string names are far too inefficient – both in terms of time and space – to tag on each IsaComponent. The compromise is to use bit vectors internally, but translate them transparently to strings for the user. This is one of the responsibilities of the TypedGraph class.

In order to do the translation, the TypedGraph class relies on the OntologyNames class which keeps a table of ontology names. This table is implemented as a simple STL vector of ontology names (as strings), where the index is the bit-offset of the IsaComponent's ontology tag bit vector.

6.3.5 Attribute Access Function Templates

Figure 21 shows several function templates: MakeAttribute(), getAttributeValue(), getValueIndirect(), and getValue(). These template functions specialize on the type of an attribute value. Recall from **Requirement 21** and the discussion in Section 6.3.1 that the type of the value in an attribute/value pair is arbitrary, but is strictly fixed when the attribute/value pair is inherited. The template functions provide type-safe access to these indeterminate types at run-time:

- MakeAttribute() is responsible for creating and associating an Attribute-subtype object with a component.
- GetAttributeValue() is more primitive than getVaue() and getValueIndirect() in that it is passed a component and an attribute name and extracts a particular attribute value

from the component, provided its type matches the expected type (an output argument).

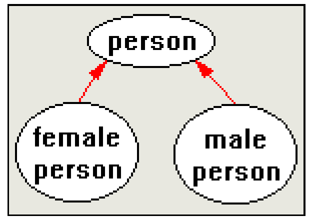
- GetValue() is passed a graph, a component ID, an attribute name, and an output type argument. It uses GetAttributeValue() to return the value if it exists and if it exactly matches the output argument type.
- GetValueIndirect() is similar to GetValue, but uses indirection in its output argument to allow the return of any subtype of the output argument type.

6.4 Using the Tool

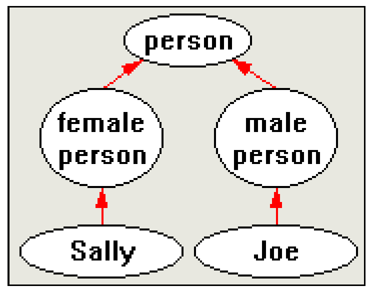
The previous sections in this chapter described the structure and design of the Constraint Graphs software. This section describes some of the pragmatic aspects of Constraint Graphs: how to use it. The next chapter will detail use of the tool further by describing how several other graphical formalisms are implemented in Constraint Graphs. In the interests of brevity, this section does not describe the details of the basic graph drawing interface; the interested reader is referred to the documentation on Constraint Graphs' predecessors (Gaines 1991b; Kremer 1993).

6.4.1 Creating a Type Lattice

In order to impose a type lattice on top of Constraint Graphs graph, it is only necessary to draw *isa* arcs among the relevant components. For example, the simple ontology of Section 4.3.3.1 is easily created by drawing *isa* arcs between *female-person* and *person*, and between *male-person* and *person*:



This world model can be just as easily extended to include some individuals:



Multiple inheritance is also allowed: one can draw more than one *isa* arc from a component. Furthermore, unlike some graph systems, arcs are first-class objects that may be the terminals of other arcs (this is illustrated in the next section). For Constraint Graphs, first-class arcs are a necessity: since *isa* arcs are subtypes of arcs, one could not create a type lattice of arcs unless arcs could terminate on other arcs.

As already described (**Requirement 19**, Section 5.11.1), attributes are inherited through the type lattice. One may edit arbitrary attributes via the attributes dialog box. For example, in the above example one might want to visually distinguish *Sally* and *Joe* as individuals instead of types by giving them rectangular surrounds and coloring them differently. An easy way of doing this to invent a new type called "individual", giving this the required visual attributes, and then having *Sally* and *Joe* multiply inherit from *individual*. The new component, *individual*, is created and assigned the appropriate attributes using the attributes dialog box as in Figure 22.

Attributes are annotated with a priority to disambiguate conflicts. In this case, *Sally* and *Joe* will inherit a shape value of "ellipse" from *person*, and a conflicting shape value of

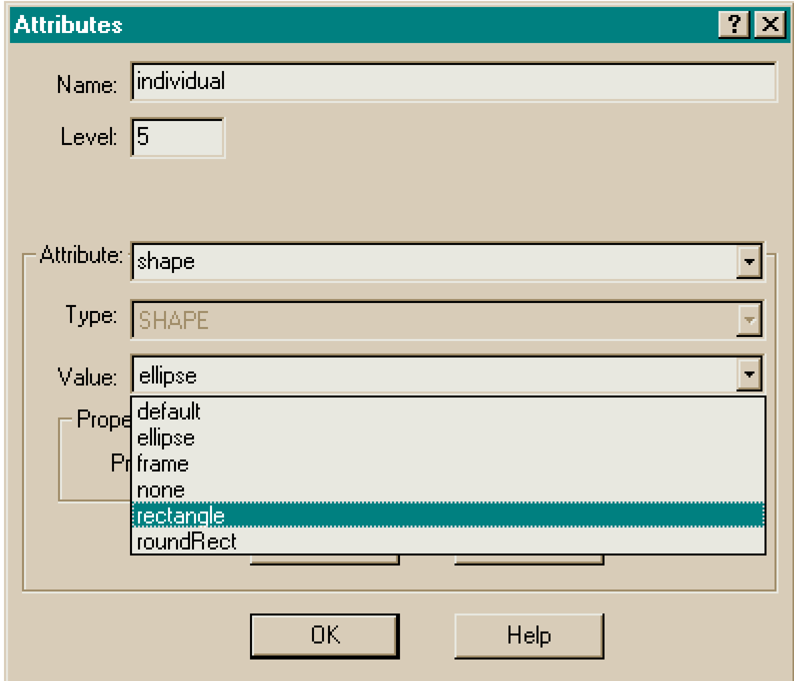
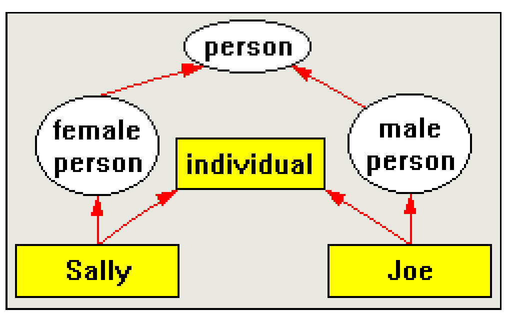


Figure 22: The Constraint Graphs attributes dialog box

"rectangle" from *individual*. One can resolve the conflict in favour of "rectangle" by assigning the shape attribute of *individual* a priority value less than that of *person*.

Finally, *isa* arcs can be drawn from *Sally* and *Joe* to *individual*:

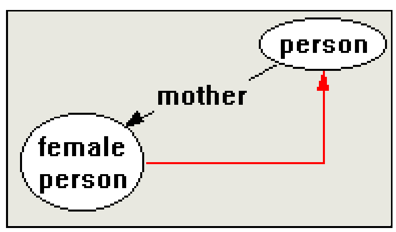


The inheritance system (Section 5.11.1) ensures that the visual attribute values of Sally and Joe are automatically changed. The observer pattern (Section 6.2) ensures that those values are automatically and immediately updated in the display.

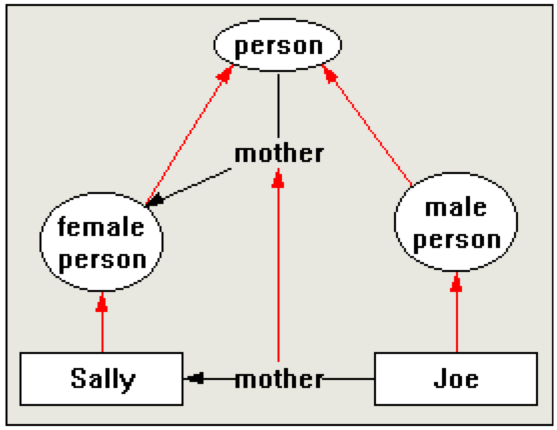
6.4.2 Constraining Arcs

Most graph-based visual language's syntax constraints revolve around constraining the component types on which arc types may terminate. Consequently, one of the fundamental operations in Constraint Graphs is restricting the component types on which arcs of particular types may terminate. This is quite easy to do in Constraint Graphs. Recall that there is no real difference between a component (object) and a type (class) in Constraint Graphs. It is therefore a simple matter of using the defining arc as the prototypical arc and connecting each of its terminals to components representing the most general legal types on which they may terminate. The ArcConformance validator (specified in Section 5.9) assures that no subtype can have a terminal on any component that is not a subtype of the component at the corresponding terminal of the supertype.

Assuming the *person* ontology of the previous section, if one wishes to create an appropriate constraint relationship called "mother", one need only draw in a *mother* relationship between *person* and *female-person*:



This automatically constrains any subtype of the *mother* arc to run between some subtype of *person* and subtype of *female-person*. For example, one might want to declare that *Sally* is the mother of *Joe*:



It would not be possible to draw the *mother* arc in the reverse direction because *Joe* is not a sub-type of *female-person*: the Constraint Graphs system would reject the transaction as shown in Figure 23.

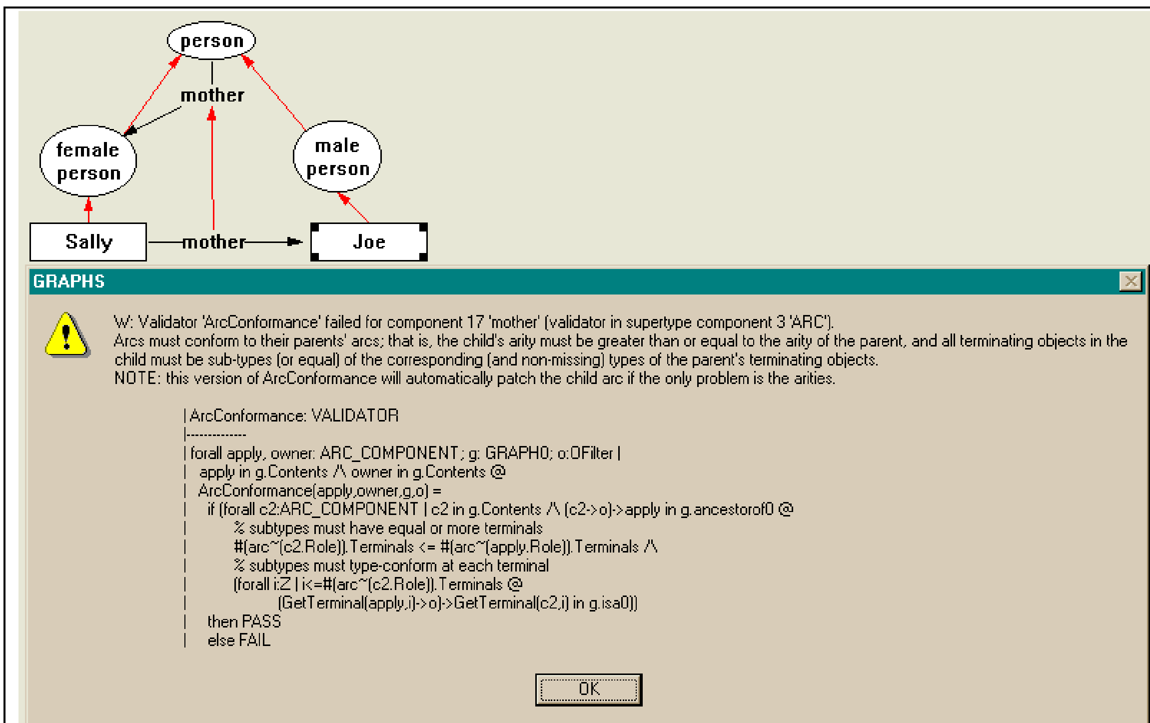


Figure 23: An attempt to draw an erroneous arc

6.4.3 Ad-hoc Constraints

Constraints on arc terminals are not the only constraints that arise in visual languages. Arbitrary constraints, such as the "first-order" constraint where no arc may terminate on another arc (described in Section 5.12), often arise. Other common "ad-hoc" constraints include:

- Arc arity constraints, where the number of terminals may be restricted to a specific number, or greater than or less than some number. Binary (arity equal to two) is by far the most common.
- Terminals of an arc may not be left dangling. (By default, terminals may be left dangling.)
- A particular component type may not participate in a multiple inheritance relationship.
- A component is considered an "object", and may *not* be a supertype to any other component.
- A component's label (or name) may be restricted to some particular syntax.

In addition, many other ad-hoc constraints (called Validators in the specification) that are used in the Constraint Graphs specification are described in Section 5.9.

It is likely not possible to express all conceivable constraints in the graphical syntax. It easy to argue that one should not attempt to: That would make the graphical syntax far too complicated to be useful. Furthermore, since future constraints may not be predictable, one would need the full expressiveness of a general-purpose programming language.

Therefore, arbitrary constraints are treated as arbitrary Boolean function objects which return false if the constraint is violated, and true otherwise. Constraint function objects can be written in the native language (C++) by a programmer and placed in Constraint Graphs' constraint library (see Section 6.5.2).

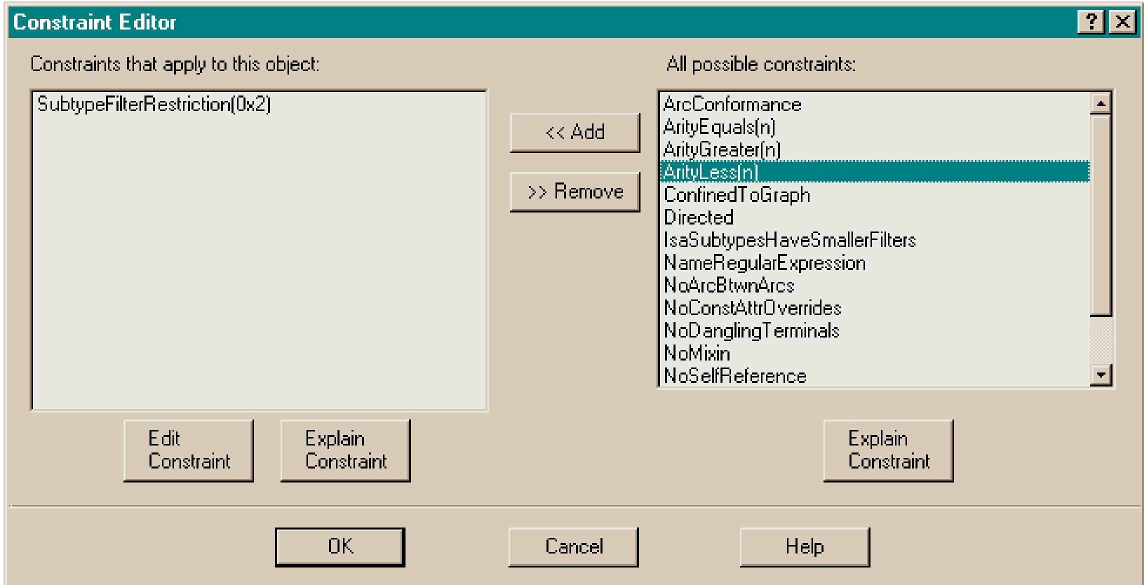


Figure 24: The Constraint Graphs constraint editor

A visual language designer (who isn't necessarily a C++ programmer) can choose constraints out of the constraint library and associate them with particular components (see Figure 24). Because the constraint function objects, like all functions objects, can have persistent state, constraints can have parameters. The selected constraint, ArityLess(n), in Figure 24 is an example: when the user presses the "Add" button, a dialog box will prompt the user for the parameter value. Each constraint may implement its own parameter dialog, so parameters may be arbitrarily complex, ranging from a simple integer parameter (for an arity constraint), to a set of regular expressions (for a label syntax constraint), to a script in a general-purpose programming language.

A constraint function object is called a Validator and has the following interface:

```
class Validator
  {
  public:
    Validator() {}
    Validator(const Validator& a) {}
    virtual bool operator()(Component0& apply,
                            Component0& owner,
                            TypedGraph& g,
                            const OFilter& f=UniversalOntology,
                            unsigned long flags=0);
    virtual Validator* clone() const;
    virtual int operator< (const Validator& r);
    virtual const string name() const;
    virtual const string specName() const;
    virtual const string doc() const;
    virtual const string spec() const;
    virtual unsigned long applicability() const;
    virtual bool primitive() const;
    virtual ostream& printOn(ostream& o) const;
    virtual istream& readFrom(istream& i);
  };
```

All validators that take parameters are known as EditableValidators. They extend the Validator interface:

```
class EditableValidator : public Validator, public EditableObject
  {
  public:
    EditableValidator();
    EditableValidator(const EditableValidator& a);
    virtual Validator* clone() const;
    virtual int edit();
  };
```

Constraints are attached to individual components and are checked whenever the component changes state. But it is not only the individual component's constraints that must be checked: both the constraints of all supertypes and of all subtypes must be checked to ensure they have not been violated. In addition, adjacent (as defined by arc connections) components must be checked, since a state change of a node may effect the legality of an attached arc and vice-versa. The actual check occurs whenever a state-

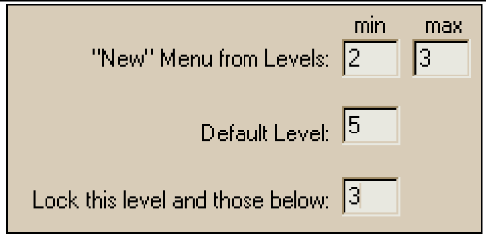


Figure 25: An excerpt from the options dialog box showing the level options

changing method is called; and if any constraint fails, the operation is undone. In this way, constraints always maintain the consistency of a graph.

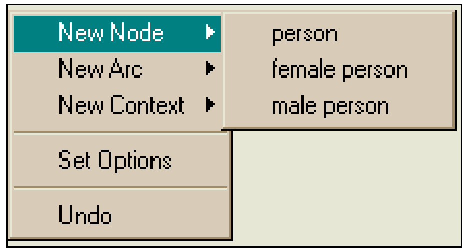
Constraints implemented in this way are reasonably efficient. They are efficient enough that they are used to implement most of the inherent conditions on constraint graphs, such as the binary, directed, and "upward" restrictions on *isa* arcs (see specification Section 5.9).

6.4.4 Using Levels

Levels are described in **Requirement 23** and in specification Section 5.1. This section illustrates some of the utility of levels. Figure 25 shows an excerpt from the options dialog box where a user may manipulate level options.

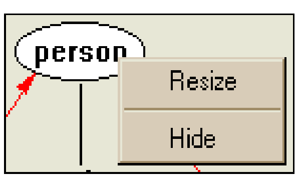
Since Constraint Graphs accepts any object as a potential type description, a user interface typically can display all of the existent components as potential initial types when the user requests creation of a new component. Clearly, this possibility is normally not what is wanted. Instead, the visual language designer would prefer to have the selection list come from a "primitive" set of potential types. In addition, the designer may wish to hide certain primitives (which are important to the system design but not relevant for the end user). Both these goals can be easily accomplished by limiting the potential creation types to a range of levels. An example case might be the *person* graph of this section, where the designer wishes to put the concepts (components) *person*, *female-*

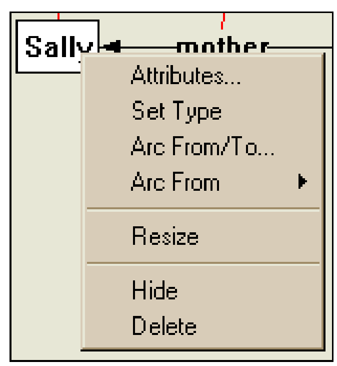
person, and *male-person* in the list of possible types, but wants to avoid doing the same for individuals (which the designer does not consider types) *Sally* and *Joe*. Thus a "new node" menu might look like this:



This can be done by setting the level of *person*, *female-person*, and *male-person* to 3, setting the level of *Sally* and *Joe* to 5, and limiting the range for "new" menus to include level 3 but not level 5 or level 1 (which includes the actual *node* type itself). The first line of Figure 25 shows how this is accomplished.

Visual language designers may also wish to lock out end users from editing fundamental constructs and accidentally disrupting the integrity of the language formalism. The following two figures show popup menus for *person* (level 3) and *Sally* (level 5) that result from level option settings shown in the last line of Figure 25 (where level 3 and below are locked):





Since *person* is a member of a locked level, it cannot be edited (except superficially), so its popup menu has been reduced to eliminate all modifying commands. On the other hand, *Sally* is not a member of locked level, so all commands are enabled on its popup menu.

6.5 Technical Details

The previous sections of this chapter discuss the structure of the Constraint Graphs implementation and the fundamentals of how it can be used to describe visual formalisms. This section is a catch-all which describes some of the more interesting technical details of the design and implementation. The topics include the flexible storage format, the flexibility and extendibility made possible by use of the librarian pattern, the cache optimization, and the hypermedia documentation.

6.5.1 Storage Format

The storage format is based on multi-part MIME (Borenstein & Freed 1993). A file (or network stream) is encoded as a simple multi-part MIME document where each part is a self-contained document describing a concept map. A primary advantage is that many concept maps may be stored or transmitted in a single file or stream. The main disadvantage is a significantly less compact format than is theoretically possible.

Just as Constraint Graphs is carefully kept independent of the interface, the Constraint Graphs data is kept separate from the interface (layout) data. This is accomplished by storing the Constraint Graphs data in its own multi-part MIME section with its own MIME type (called "application/x-ConstraintGraphs"), and the interface data in a separate MIME part (called "application/x-CMap"). See Figure 26. This scheme allows a great deal of flexibility, since not only may multiple x-ConstraintGraph/x-CMap pairs be stored in the same file, but also several separate x-CMap parts may follow a single x-ConstraintGraph part, which corresponds to several views of a single Constraint Graph. In addition, the multi-part MIME format allows for annotation with comments and a huge

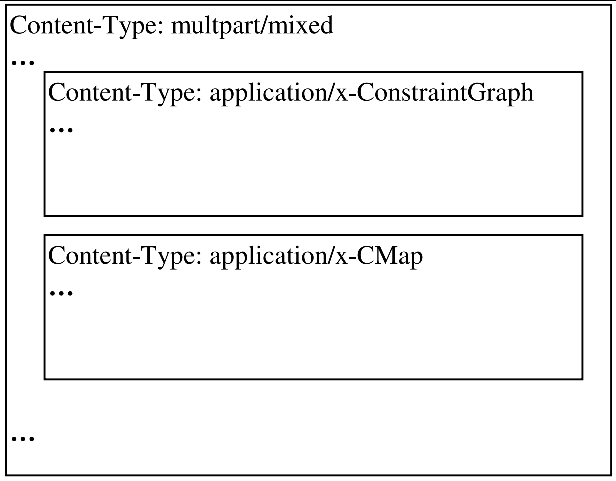


Figure 26: A schematic view of the Constraint Graphs storage format

variety of multimedia parts without disturbing the simple layout. (A Constraint Graphs program may simply ignore MIME parts for MIME types it does not understand.)

The multi-part MIME format also makes it easy for other programs to read CMap data up to the level that the "foreign" program may understand. The simplest case in point is KSIMapper (Kremer & Gaines 1996) that reads x-CMap data as its native storage format. It can easily read a ConstraintGraph file just by ignoring the x-ConstraintGraph MIME parts. KSIMapper cannot understand the x-ConstraintGraphs part, but there is enough information in the x-CMap part to allow it to do a very good job of displaying the data. Programs that read part of the data in this way should not be able to edit (and write back) the data as they cannot know of interdependencies within the data.

The reader may wonder how the x-CMap interface can infer the individual associations between components in the interface and components in the Constraint Graph. The answer lies with the x-CMap format: the x-CMap format allows each component to be tagged with an extension that may be ignored if not understood by the reading application. In this case the extension is merely the identification number of the

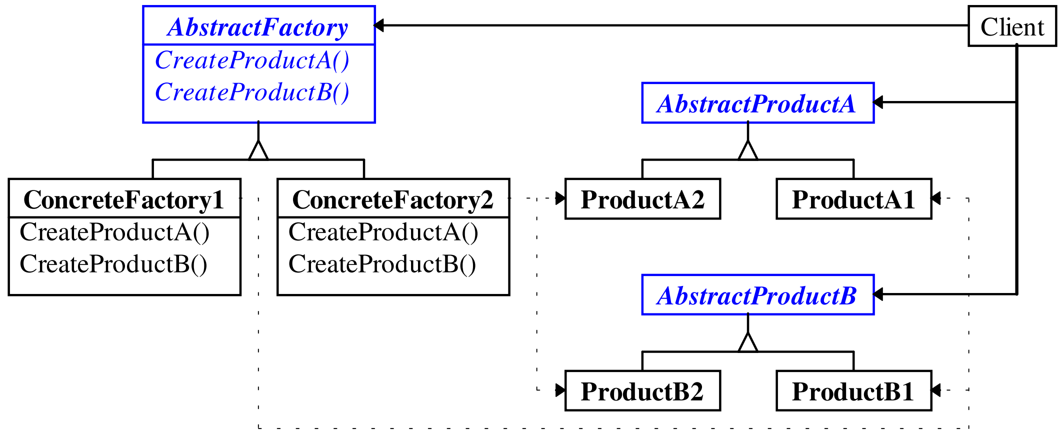


Figure 27: The abstract factory pattern (Gamma et al. 1995)

corresponding component in the Constraint Graph. See Kremer & Flores-Mendez (1996) for a complete explanation.

6.5.2 Flexibility and Extendibility: the Librarian Pattern

The Constraint Graphs software is not only a stand-alone program, but is also a framework (Pree 1995) that may be easily extended to implement other, unforeseen, applications. As such, it must be extremely flexible with respect to the exact classes it handles. The design restricts the superclass of all Constraint Graphs (and CMap) objects, but the layered application may dictate the exact classes (types) of almost all of the objects. Thus, a layered application can modify and extend object behaviour and functionality.

An obvious way to achieve this flexibility is by use of the abstract factory pattern as shown in Figure 27 (Gamma et al. 1995). Abstract factory is used in several places in the Constraint Graphs implementation. However, the abstract factory pattern relies on concrete classes being available at compile time; whereas there are several cases in Constraint Graphs where the exact type of an object to be created is not known until run time. Examples include:

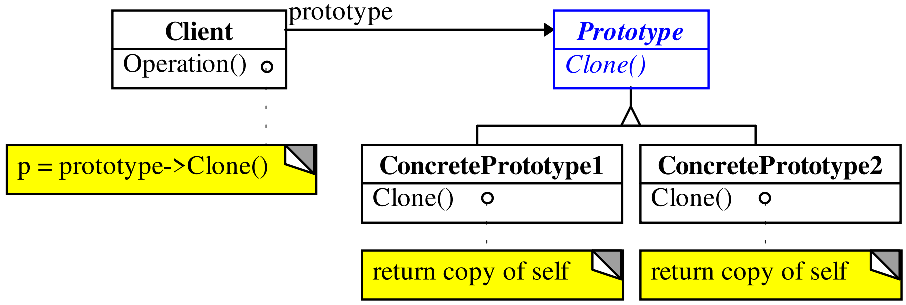


Figure 28: The prototype pattern (Gamma et al. 1995)

- The exact type of a Constraint Graph component is not known until it is read in from a file or a stream at run time. The name of the exact class is read in with each component's record.
- A similar situation arises when both CMap's behavioural and visual graphics objects (see Section 6.2) are read in from a file or stream.
- A CMap behavioural graphic may be required to change the type of its delegate visual graphic object (i.e.: it must discard the visual object in favour of a new visual object) at run time. Typically, this event happens in response to the user selecting a new type in a dialog box. The new visual object must be dynamically constructed based on the text string supplied by the selection.

This problem is addressed by a new pattern called the library pattern.

The library pattern is an extension of the prototype pattern (see Figure 28) documented by Gamma et al (Gamma et al. 1995) and is used extensively in the Constraint Graphs implementation. The library pattern extends the prototype pattern by replacing the client's direct reference to the prototype with a container (ObjectLibrary) which acts as an intermediary (see Figure 29). The ObjectLibrary container holds any number of prototypes which are indexed by string names. Instead of the client directly cloning a prototype, it asks the ObjectLibrary to do the cloning on its behalf, passing a string

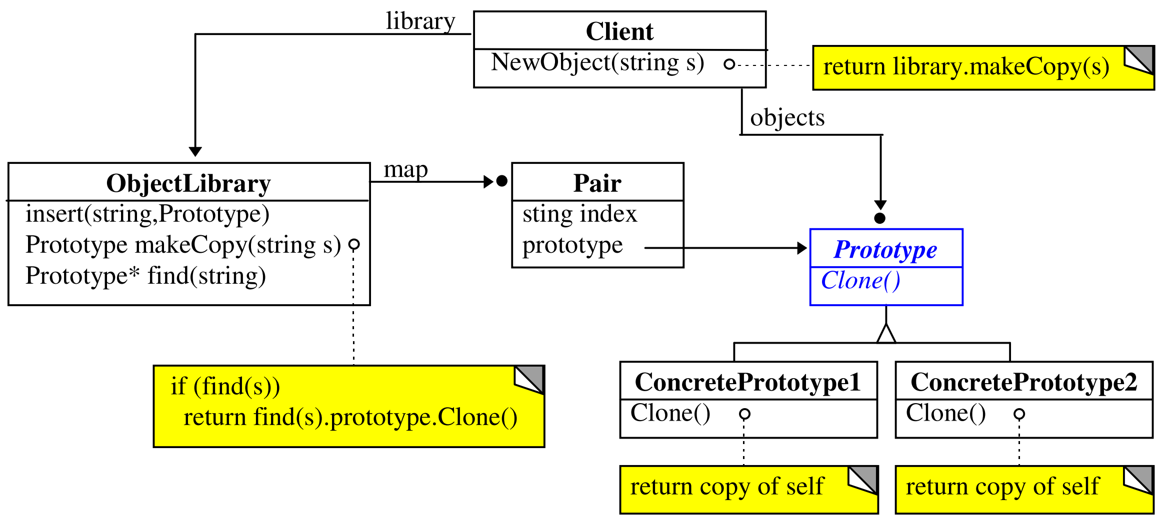


Figure 29: The library pattern

parameter to specify the prototype object it needs. It is easy for the implementation to initialize the ObjectLibrary's contents at run time.

One danger with using the library pattern as shown in Figure 29 is that the client may expect the returned object to conform to a specific type specification. But there is nothing in the figure to restrict the type of the cloning object (beyond being a Prototype). In order to ensure type safety for clients using ObjectLibraries, the library pattern is implemented as C++ template (see Figure 30). Thus, a client using an ObjectLibrary of type ObjectLibrary<VisualGraphic> can always be guaranteed to retrieve an object of type VisualGraphic*.

6.5.3 Optimization: Caching

An interesting aspect of implementing Constraint Graphs from the specification is the structural differences between the specification and the implementation. There are, of course, differences that spring from the inclusion structuring of the Z specification

```
template <class T, int hasDefault=1>
class ObjectLibrary : public map<string,T*,less<string> >
  {
  public:
    T* makeCopy(const string& x) {
      T* ret = access(x);
      if (ret) ret = ret->clone();
      return ret;}
    T* access(const string& x) {
      iterator i;
      T* ret = NULL;
      if ((i=find(x))!=end())
        ret = (*i).second;
      else if (hasDefault && (i=find("default"))!=end())
        ret = (*i).second;
      return ret;}
    pair<iterator,bool> insert(const string& x, T* t) {
      return map<string,T*,
        less<string> >::insert(pair<const string, T* >(x,t));}
    pair<iterator,bool> insert(T* t) {
      return map<string,T*, less<string> >::insert(
           pair<const string, T* >(typeid(*t).name(),t));}
  };
```

Figure 30: The library pattern template

language versus the object-oriented inheritance structuring of the C++ programming language. This is detailed in the introduction to chapter 5. But there are also efficiency considerations which dictate style differences between the specification and the implementation. Unfortunately, these differences tend to cloud what should otherwise be a tidy one-to-one correspondence between the specification structures and implementation structures. Nonetheless, these differences cannot be avoided.

As a case in point, the algebraic style of the specification is easily translated to C++ code, but leads to extremely inefficient C++ code. For example, the specification of the operator *ancestorof* is specified as

```
forall c,p:Contents; o:OFilter @
   (p->o) ancestorof c <=> paths_btwn(c,p,o) /= {};
```

which would lead to some fairly inefficient C++ code: A direct translation would be a double-nested *for* loop iterating the entire contents of the graph n^2 times (assuming n is the number of component in the graph). Furthermore, the code would compute paths_btwn() n^2 times; since the literal translation of paths_btwn() is itself an polynomial function, the time complexity of *ancestorof* is very high.

An initial implementation which factored out paths_btwn(), but used the double-nested *for* loop, took close to a full second to compute the ancestorof() function for a reasonably large graph. An optimization which cached the *parentof* relationship as local pointers (and used this relationship instead of paths_btwn()) could perform the same operation on the same graph in less than 0.001 second.

This dramatic increase in efficiency comes at the cost of corresponding alienation between the specification and implementation. As the appropriate optimizations are applied to the implementation, there comes to be a less and less clear correspondence between the specification and implementation. Thus, the formal specification gradually becomes less and less useful as a documentation source or as guide to code maintenance. This problem can be partially mitigated by the use of hypermedia documentation, described in the next section.

6.5.4 Hypermedia Documentation

The previous section discusses the problem of alienation between the specification and implementation; this section explores at least a partial solution. The Constraint Graphs specification is written in the form of an HTML document as described in the introduction to chapter 5. To enhance the correspondence between specification and implementation, each part of the specification is tagged with a hypertext anchor. HTML markup tags are added in comments in the C++ source code and header files of the implementation. Thus, both the specification and implementation files can easily be given hypertext links that link to the corresponding passages in the implementation or specification. For example, the implementation of the TypedGraph class is tagged as follows:

```
/*<a name="classTypedGraph">
  <a href="GraphsZ.htm#GRAPH0">Z spec (GRAPH0)</a>,
  <a href="GraphsZ.htm#TYPED_GRAPH">Z spec (TYPED_GRAPH)</a>,
  <a href="graphs_cpp.html#classTypedGraph">implementation</a> */
class TypedGraph : public RefCountingObject
  { . . .
```

The code may then be converted into HTML documents (which only involves copying them to files with extension ".html" since the HTML markup tags are already included as comments) and the specification and C++ files refer to one another. This makes it easy for programmers to browse the specification, then look at the corresponding C++ code quickly and easy by following a hypertext link. Likewise, if programmers use WWW browsers to browse the C++ code, they can quickly and easily retrieve the full specification in context. The reader may be interested in seeing this documentation at

http://www.cpsc.ucalgary.ca/~kremer/graphs/GraphsZ.htm

6.6 Summary

This chapter has provided an overview of the Constraint Graphs program design and implementation. The most important point is the clear separation between the abstract Constraint Graph and the user interface – the visualization of the Constraint Graph. Section 6.2 describes the CMap library that acts as the interface and Section 6.3 describes the class library design for Constraint Graphs itself.

Section 6.4 is more pragmatic and describes the actual use of the Constraint Graphs program: how to create a type lattice within the Constraint Graph, how to constrain the types of the objects at arc terminals, how to use ad-hoc constraints, and how to use ontological levels to enhance the user interface for an end user.

Section 6.5 picks up on some of the more interesting technical details including:

- the multi-part MIME storage format, which separates Constraint Graph concerns from user interface concerns
- the library pattern used in the program, which is used extensively to enhance flexibility and extendibility.
- caching of arc relationships, which is a dramatic speed optimization over the "obvious" implementation from the specification
- hypertext documentation, which helps to keep the association between the parts specification and the corresponding parts of the implementation.

This chapter serves as documentation of *one* possible design based on the specification given in Chapter 5. There are always many possible designs that may arise from a good (not over-specified) specification. This design illustrates the many tradeoffs between conflicting "soft" goals such as clear conformity to the specification, easy-to-understand code, run-time efficiency, modularity, modifiability, flexibility, reuse of pre-existing components, and production of reusable components. For example, the design placed higher priority on clear conformity to the specification than on easy-to-understand code, which resulted in a very clear correspondence between specification and implementation, but less-than-desirable class hierarchy where unnecessary C++ classes were introduced to match the (necessary) corresponding Z schemata (as discussed in Section 6.3). In another example, the design placed higher priority on clear conformity to the specification and easy-to-understand code than on run-time efficiency, but this had to be compromised when it was found that the frequent following of *isa* arcs in the style of the specification lead to unacceptably slow run-time performance. Caching of *isa* arc references had to be added to alleviate the problem (Section 6.5.3).

On the other hand, reuse of well-designed pre-existing components took place without compromise both at the design and at the implementation levels. Several of Gamma et al's (1995) patterns were reused in the design which served to clarify the design and help document the code. The C++ Standard Template Library was also reused extensively without any compromise: the STL components are very close models to the constructs of the Z notation, and so translate almost exactly while still maintaining impressive efficiency and type safety. For example, the STL *set* template class corresponds to the Z *set*, and the STL *vector, dequeue*, and *list* template classes all corresponds to the Z *seq* (sequence).

The documentation of the design given here is not complete, but illustrates most of the major decisions. It should be obvious that the design touches on many areas that are not within the domain of the specification. It has already been stated that the specification does not cover the user interface, but there are many other areas that the specification purposefully does not cover but do appear in the design. For instance, the specification should not (and does not) cover the storage format, or even *if* the data should be stored; but this topic is covered in the design (Section 6.5.1). The specification defines *what* a concept map is (in terms of abstract data structures and operations), not *how* it is implemented or the specific data formats. Details such as storage formats are merely design details (unless storage format is what is being specified – which it isn't), and so these decisions are deferred from specification to design.

Chapter 7

Some Visual Languages Implemented in Constraint Graphs

The previous chapter describes the Constraint Graphs program from a design point of view. In contrast, this chapter is far more pragmatic and describes how to use Constraint Graphs to implement several formal and semi-formal visual languages.

The chapter starts with a very simple, semi-formal language, gIBIS (Graphical Issue Based Information System) (Conklin & Begeman 1987). The description of gIBIS serves to introduce the reader to the basic functions of Constraint Graphs. But the gIBIS description does not exercise the system's full capabilities. The more advanced features of Constraint Graphs are illustrated in Section 7.2, which presents an implementation of an extension to Conceptual Graphs (Sowa 1984). Conceptual Graphs has many uses in knowledge representation and natural language representation and has a reasonably large and active user community. Finally, Section 7.3 presents KDraw (Gaines 1991b), which is a visual form of the CLASSIC (Borgida et al. 1989) knowledge representation language. This language presents a real challenge for Constraint Graphs, serving to exercise most of its power and showing some very interesting problems in visual language representation.

7.1 gIBIS

gIBIS (Conklin & Begeman 1987) is an acronym for "Graphical Issue-Based Information System". It is a simple visual language used in business decision making. There are several versions, and all are very simple. The language consists of typed nodes and typed

Table 3: gIBIS arc types, showing the restrictions on the node types at the terminals.

Arc Type	Source Node Type	Sink Node Type
questions	Issue	(any)
is-suggested-by	Issue	(any)
specializes	Issue	Issue
generalizes	Issue	Issue
responds-to	Position	Issue
supports	Argument	Position
objects-to	Argument	Position

arcs only (there are no contexts). The version used in this example has only three node types:

- *Issue*: usually a decision to be made
- *Position*: a statement of a possible resolution of a particular Issue
- *Argument*: a statement supporting or objecting to a particular Position

There are also seven arc types. All of them are binary, directed arcs and are limited to connecting only certain node types to certain node types. Table 3 lists the arc types and includes the arc terminal restrictions.

Figure 31 shows a simple gIBIS map, which may be created as part of making a decision about which type of computer an organization is going to buy. In this map, Issue nodes are shown as ellipses, Position nodes are shown as rounded rectangles and Argument nodes are shown as rectangles.

In order to create a Constraint Graph system to support the construction of gIBIS maps, one needs only:

1. Limit the types of nodes to Issue, Position, and Argument.
2. Limit the types of arcs to questions, is-suggested-by, specializes, generalizes, responds-to, supports, and objects-to.
3. For each arc type, restrict the end points to terminate on only the legal node types as per Table 3.

All of this is easily done using Constraint Graphs. The following discussion will follow the reasoning that a visual language designer might go through when constructing a Constraint Graphs gIBIS implementation.

7.1.1 The Top-Level gIBIS Node and Arc

Since Table 3 shows that the *questions* and *is-suggested-by* arcs may terminate on any of the three node types, it makes sense to capture the notion of a general gIBIS node (the union of *Issue, Position*, and *Argument*). In addition, one might want to capture the

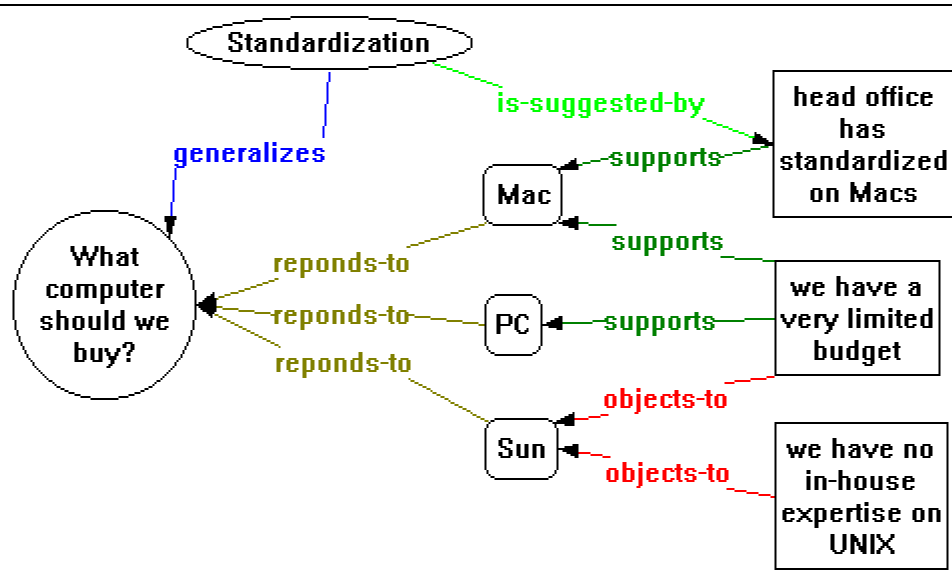
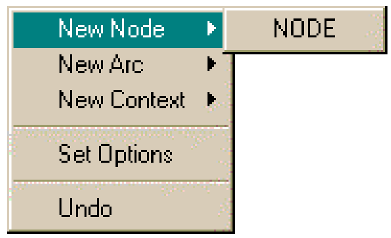
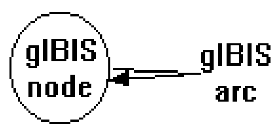


Figure 31: A simple gIBIS discussion about buying a computer

notion of a general gIBIS arc, and specify that it must connect a gIBIS node to another gIBIS node. (While the need for a "gIBIS arc" might seem speculative at this point, good reasons for choosing to implement it will become apparent near the end of this discussion.) The gIBIS node is created by left-clicking on the background and choosing New Node|NODE from the popup menu:



After naming the node gIBIS node, the gIBIS arc may be created in a similar way and both arc terminals may be attached to the gIBIS node. This yields the following map:



Both terminals of the *gIBIS arc* terminating on the *gIBIS node* serve to restrict the terminals of any *gIBIS arc* subtype to terminate on components that are subtypes of *gIBIS node* (see Section 6.4.2).

The visual attributes of the gIBIS node and the gIBIS arc are specified using the attributes dialog box. The attributes dialog box for the gIBIS node is shown in Figure 32. The *Name* box lets the user type in the name of the node. The *Level* box lets the user specify the level of the component (see Section 6.4.4). Here, level 2 is chosen because "gIBIS node" is not a legal gIBIS node type – it is just a necessary part of the ontology, and must be hidden from the end user. At this early point, the designer realizes certain components must be hidden from the end user, so it is appropriate to choose the leveling structure. A typical breakdown is as follows:

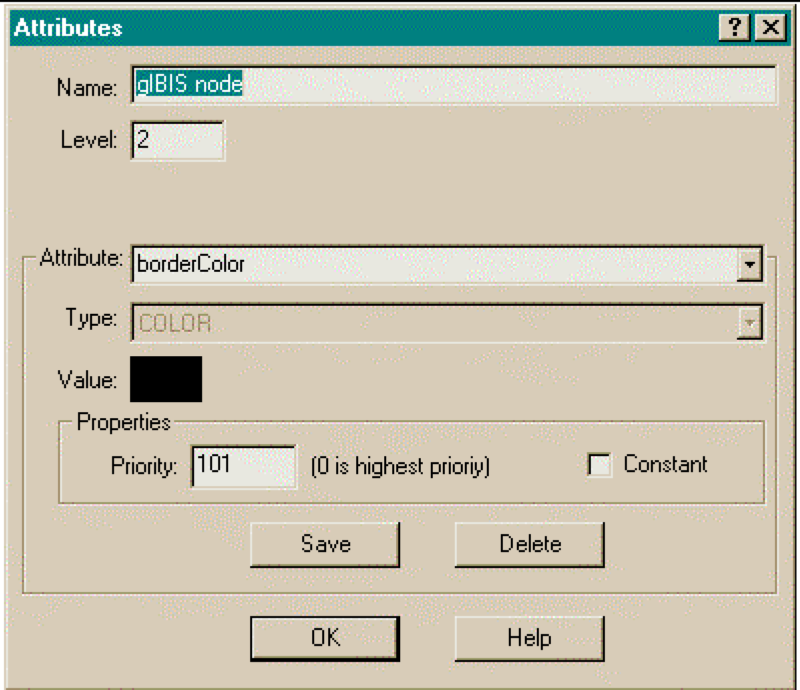


Figure 32: The attributes dialog box for "gIBIS node"

- Level 1 is the Constraint Graphs system level. Components belonging to this level are not normally edited by either the visual language designer or the end user.

- Level 2 is the hidden part of the visual language design. Components belonging to this level are created and edited by the visual language designer, but the end user is not normally aware of their presence because they don't show up anywhere in the user interface.

- Level 3 is the "public" part of the visual language design. Components belonging to this level are created and edited by the visual language designer and are visible to the end user, but they are typically locked from editing by the end user (since editing these component may disrupt the integrity of the visual language).

- Higher numbered levels are usually the domain of the end user. Components belonging to these levels are created and edited by the end user. The components in these levels are almost always subtypes of components in level 3. The default component level for the end user is usually set at level 5.

This level assignment suffices for the current purposes. Both the general gIBIS node and the general gIBIS arc are assigned to level 2. The other, concrete, gIBIS types can be assigned to level 3.

The attribute box of Figure 32 is where the user can specify attribute/value pairs. While the user may specify arbitrary attribute names, the following attributes are used by the system for nodes:

- *borderColor*: type COLOR (24 bit hexidecimal value in RRGGBB format).
- *borderStyle*: type UNSIGNED LONG INTEGER (maps to dotted, dashed, etc.)
- *borderWidth*: type INTEGER (pixels)
- *constraints*: type CONSTRAINT (an editable list of constraints)
- *fillColor*: type COLOR
- *shape*: type SHAPE (ellipse, frame, none, rectangle, roundedRect…)
- *textColor*: type COLOR

All these attributes are set for gIBIS node except for *constraints*, which is left empty.

The attributes for arcs include all of those for nodes, plus

- *lineColor*: type COLOR
- *lineStyle*: type UNSIGNED LONG INTEGER (maps to dotted, dashed, etc.)
- *lineWidth*: type INTEGER (pixels)
- *connector*: type CONNECTOR (describes the type of connector: bidirected, bidirected labeled, directed, directed labeled, plain, plain labeled…)

The attributes are all specified for the general gIBIS arc.

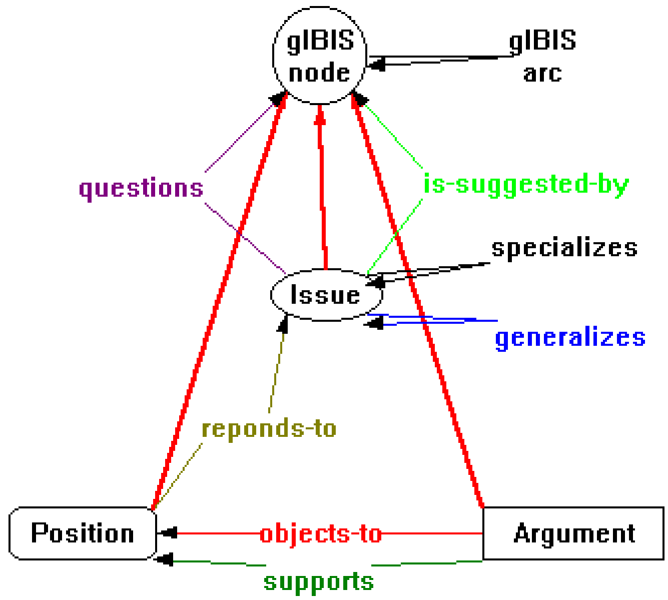
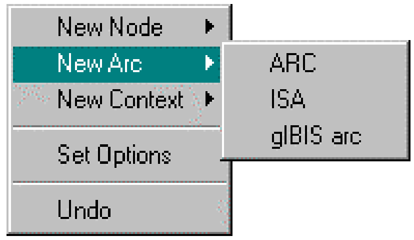


Figure 33: The Constraint Graphs definition of the gIBIS model

7.1.2 Defining the Rest of the gIBIS Nodes and Arcs

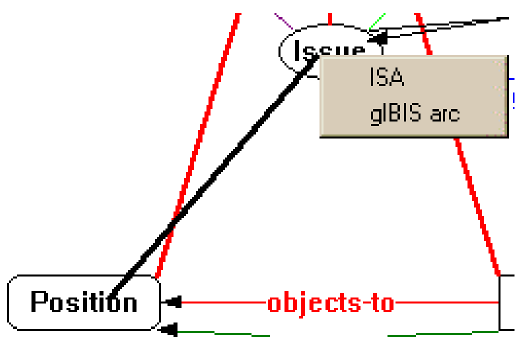
Finally, the actual gIBIS nodes and arcs can be defined. The three gIBIS nodes, *Issue, Position,* and *Argument*, are first defined as top-level Constraint Graph nodes, then an *isa* arc is drawn from each to the general gIBIS node. Immediately after the *isa* arcs are drawn, the visual attributes (colors, shape) of the nodes changes to match those of the general gIBIS node. But *Position* and *Argument* are meant to have different shapes, so the attributes dialog for these two nodes is invoked to change the *shape* attribute appropriately. These operations yield the nodes in Figure 33.

The arcs in Figure 33 are created in a similar way, but to actually show the *isa* arcs in the figure would clutter up the diagram unreasonably, so a slightly different technique is used. Instead of creating the arcs as generic Constraint Graph arcs, they are created directly as subtypes of the general gIBIS arc. This is done via the background popup menu

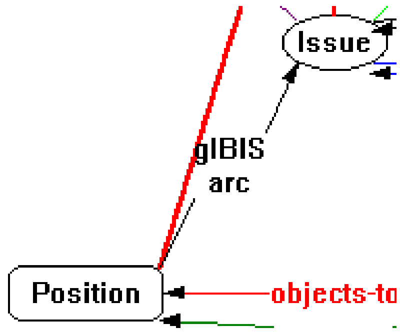


by selecting "gIBIS arc" from the "New Arc" sub-menu. This has the effect of creating the appropriate *isa* arc in the Constraint Graph, but not drawing the *isa* arc in the interface. After each arc is created in this way, the end-points are merely dragged and dropped on the relevant nodes, and then the attributes are appropriately modified to complete the definition shown in Figure 33.

An alternative approach to creating a *gIBIS arc* subtype without a visible *isa* arc is to select "Arc From/To…" from the node popup menu, which starts an elastic line selection anchored on the node. For example, to create the *responds-to* arc of Figure 33, the user selects "Arc From/To…" from *Position*'s popup menu. This creates an elastic line anchored on the *Position* node. The user then drops the elastic line over the *Issue* node. The system will automatically infer all legal arc types between the nodes and display the possibilities as a menu:



Since a *gIBIS arc* subtype is desired, it is selected; and the system will automatically draw a *gIBIS arc* subtype between the two nodes:



The name of the arc is copied from *gIBIS arc*, but the arc can be edited to change its name and any other desired attributes. In this case the arc will be renamed *responds-to*.

All of the gIBIS components, except for "gIBIS node" and "gIBIS arc" (which are level 2) should be placed in level 3, since level 3 components will be made visible to the end user. The level 2 components will be hidden.

7.1.3 Preventing Further Subtyping

The gIBIS visual language does not include any notion of subtyping and inheritance. Therefore, end users should be prevented from subtyping any of the nodes or arcs they create. (They must subtype, or instantiate, the basic gIBIS types, but they must not create any subtypes of the instantiated components.) Constraint Graphs provides the functionality to do this through the "ParentOfIndividual" constraint, which is added to each of the level 3 gIBIS components.

Constraints are just attributes in this Constraint Graphs implementation, so the attribute dialog is invoked for each component and the constraint attribute is selected. The "value:" portion of the attributes dialog is an edit button. Pushing the edit button brings up a special constraint editor shown in Figure 34.

The left-hand pane in Figure 34 contains constraints that are currently applied to the component, and the right-hand pane contains all the applicable constraints from the

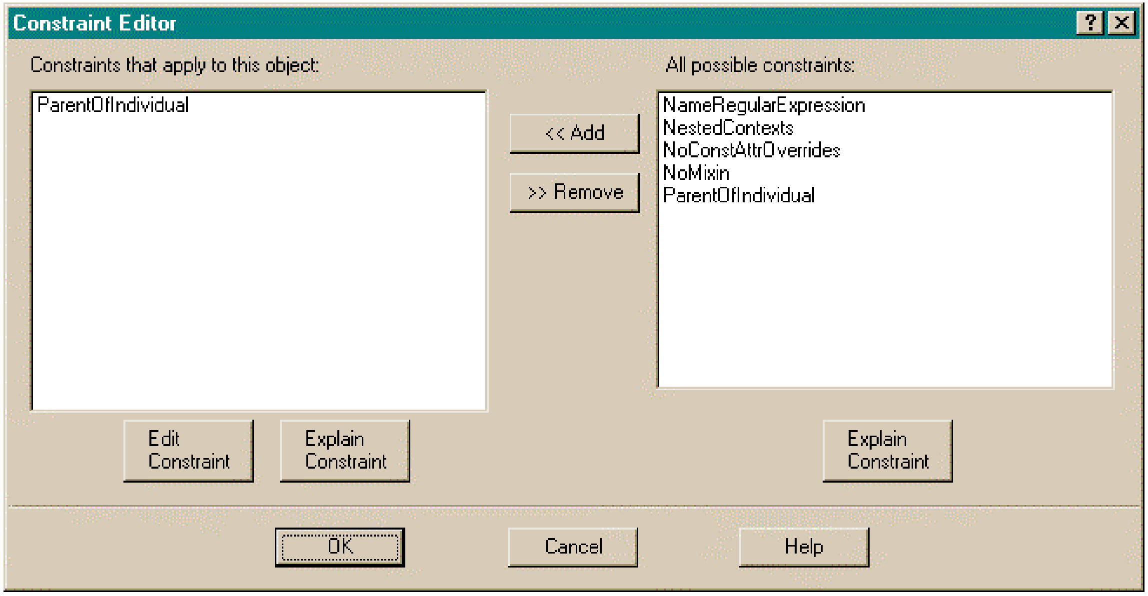


Figure 34: The constraint editor, specialized for nodes (arc editors have a longer list of "possible constraints").

current constraint library. (Nodes, arcs, *isa* arcs, and contexts all may have different sets of possible constraints.) The "Explain Constraint" buttons are used to get explanations, in both English and in Z, of the meaning of the constraint highlighted in the panes. The "Edit Constraint" button is used to change the parameters of a constraint via yet another dialog.

In the gIBIS definition, tagging each level 3 gIBIS component with the ParentOfIndividual constraint will prevent the end user from being able to subclass or instantiate from any sub-gIBIS component.

7.1.4 Hiding the Details from the End User

All that remains to do to create the Constraint Graphs gIBIS model is to hide the unnecessary details from the end user. This is done by using the levels options from the Constraint Graphs options dialog as shown in Figure 35. The "New Menu from Levels"

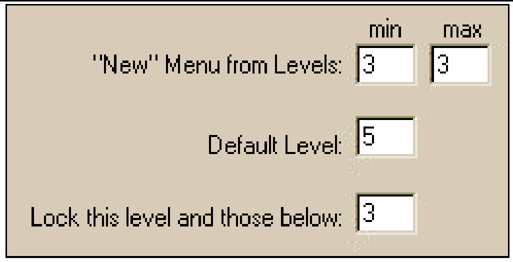


Figure 35: The Constraint Graphs levels options

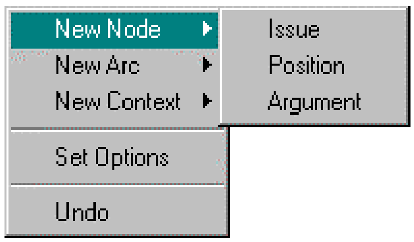
fields select the levels from which the system will infer types. The types in these levels will be the only ones used in the background's (and components') popup menus' "new" sub-menus. They will also be the only types considered when the system must infer the types of arcs (see Section 7.1.2 for an example). Setting the minimum level to 3 prevents the end user from seeing the Constraint Graph base types (level 1) and the gIBIS base types (level 2). Setting the maximum level to 3 prevents the end-user created components (which can only be levels 4 and above, see below) from being displayed in menus or used in type inferences.

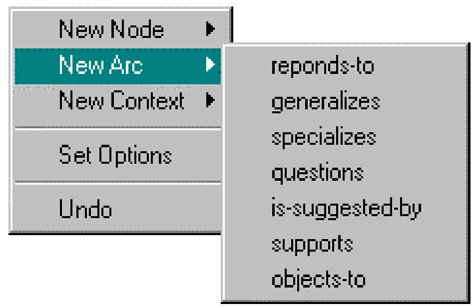
The "Default Level" field in Figure 35 is used to supply a default for the level field of the attributes dialog (Figure 32).

The "Lock this level and those below" field in Figure 35 is used to prevent the end user from editing, deleting, or creating new components in the specified levels. This prevents the user from tampering with the visual language definition, and perhaps inadvertently corrupting the integrity of the language. Components within the locked levels automatically have truncated popup menus: the editing and deletion operations are dropped.

7.1.5 The Interface for the End User

The end user's environment, after all the above is complete, is specialized to the gIBIS semi-formalism. The background menu's "New Node" and "New Arc" submenus are limited to just the gIBIS node types:





Since there are no contexts in gIBIS, the "New Contexts" submenu is empty. Constraint Graphs allows the creation of new arcs in isolation (unanchored to any nodes at all). These are displayed in the obvious manner:

Any component may be selected by clicking on it, which displays its handles:

The handles at the terminals may be dragged and dropped on another component to anchor it on that component. But if the terminal is not legal according to its constraints, an error message is displayed and the operation is undone. For example, if the "to" (arrowhead) side is dragged over a *Position*-type arc (which is not legal, according to the definition), the connection is refused, and an error message is displayed (see Figure 36).

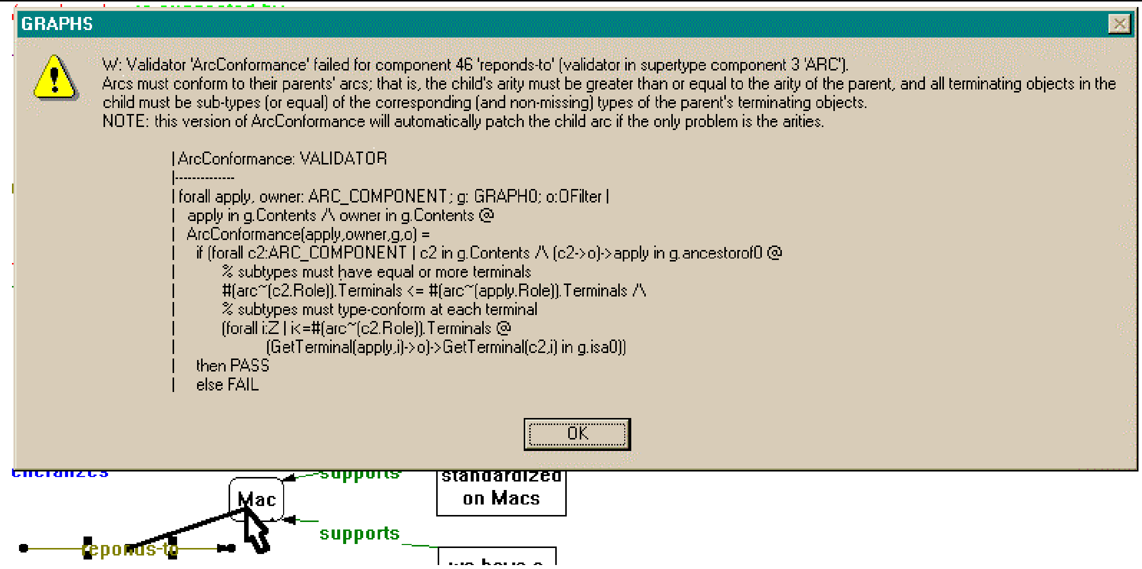
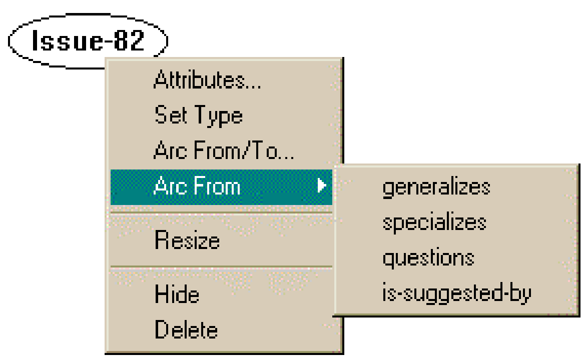


Figure 36: A refused arc connection.

There are also two other ways to create arcs. Both are via the node popup menu:



The "Arc From" submenu creates a specific-typed menu rooted at the current node (with the other terminal(s) free. This sub-menu contains only the arc types that may legally be rooted at the target node. In this case, there are only four possibilities.

An arc may also be created by clicking on the "Arc From/To…" menu item, then dragging an elastic line from the current node to another node. The system will infer any legal arc types between the two nodes according to the definition. If there is a unique arc type, an arc of that type will be drawn between the two nodes. If there is more than one

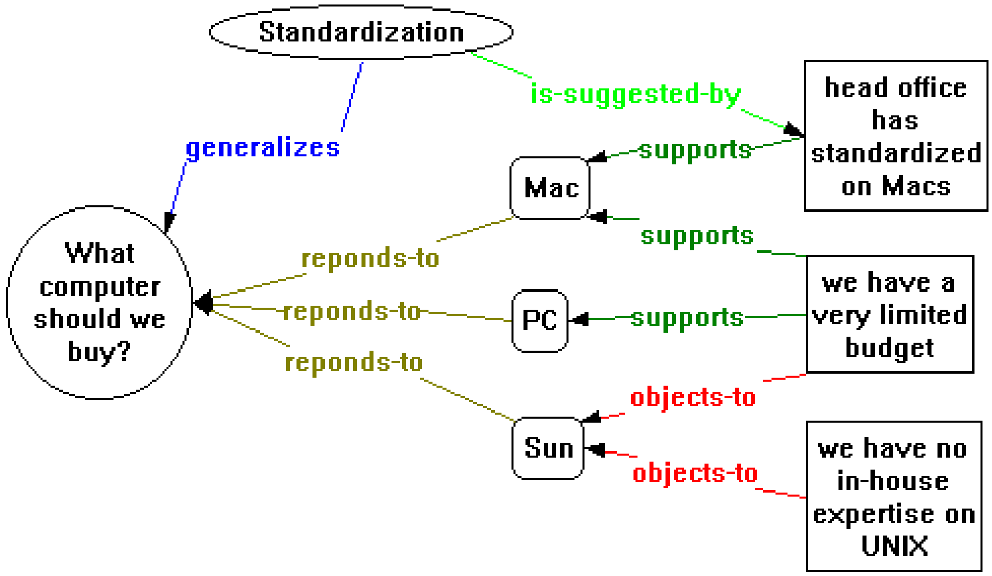


Figure 37: An example concept map created with the Constraint Graphs definition of gIBIS

possibility, the system will use a menu to query the user for the chosen type, and the arc will be drawn. If there are no legal arc types between the two nodes, an error message will be displayed.

Using these techniques, a gIBIS concept map may easily be created. Figure 37 is an example. Other than dangling arcs, it is not possible to create a syntactically incorrect gIBIS map. If the visual language designer wanted to avoid dangling arcs, then the ad-hoc constraint "noDanglingTerminals" (which is resident in the constraint library) could have been added to the constraints for the "gIBIS arc" arc.

It is now possible to explain more of the rationale behind creating the hidden "gIBIS arc" arc in the first place. Arc-global constraints, such as the "noDanglingTerminals" constraint from the previous paragraph, may be easily added using the "gIBIS arc" arc. Furthermore, other arc-global attributes may be added at "gIBIS arc". For example, if users become very familiar with the arc types, they may be fluent in recognizing the arc types by their color: the arc labels would become quite redundant. Since all the arcs used in the concept map inherit their "connector=directed labeled" attribute from "gIBIS arc", one only need change the connector attribute to "connector=directed" to erase all labels in the concept map. This makes for a far less cluttered concept map (compare Figure 38 to Figure 37). The labels may all be restored by merely changing the one attribute back to its original value.

Section 7.1 has described the implementation of the gIBIS concept mapping language in detail. The gIBIS language acts as test of Constraint Graphs theory and framework in that gIBIS was not originally considered during the development of the theory and design of the framework. gIBIS is a fairly simple and straight-forward concept mapping language; it, therefore, makes a good first example. But its implementation presents no challenge to the framework. Unlike some of the other languages described in this chapter, it caused no iterations back to the early design phases of the project. The other language implementations are presented in much less detail, but serve to illustrate some of the

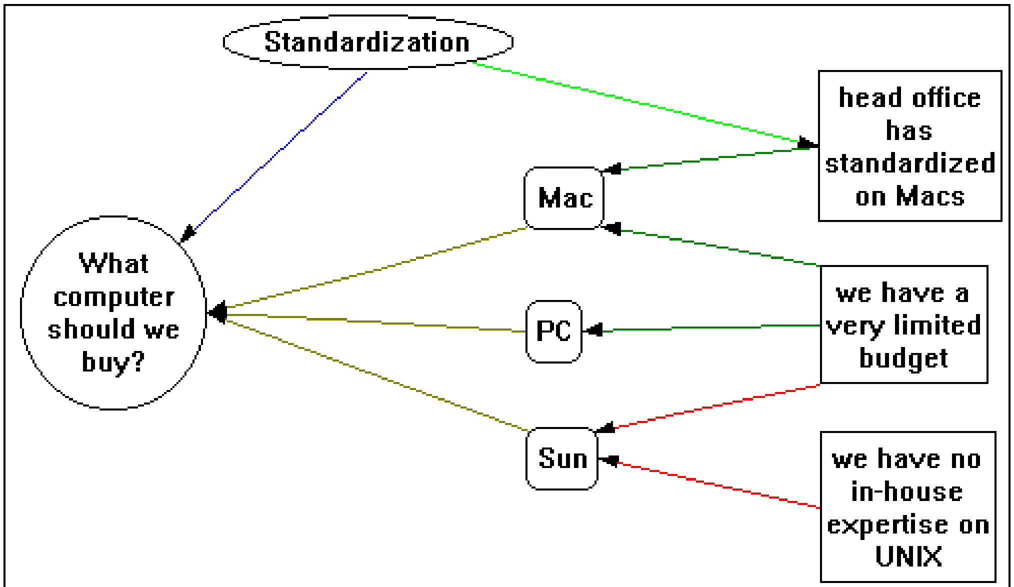


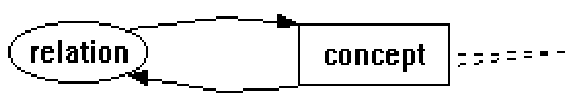
Figure 38: The same concept map as Figure 37 using unlabeled connectors.

more obscure features of the framework.

7.2 Conceptual Graphs

The syntax for Sowa's Conceptual Graphs is implemented in a straight-forward manner in Constraint Graphs. However, there are two major interpretations that can be applied. Both are logically equivalent, but the less traditional interpretation allows Constraint Graphs to more conveniently capture some of the semantics of the Conceptual Graphs language.

The traditional interpretation is that Conceptual Graphs are bipartite graphs consisting of *concept* nodes and *relation* nodes that may be interconnected by directed arcs. Thus, there are only two binary, directed, non-subtype-able arc types: *relation-to-concept* and *concept-to-relation*; these support the bipartite nature of the system. In addition there exists a third, binary, undirected arc type, called *coreference*, which may only occur between two *concept* nodes. This interpretation is easily modeled in Constraint Graphs (*context* nodes are omitted here):



However, an alternate interpretation is that relations are arcs (labeled arcs with an elliptical surround). This interpretation dispenses with *relation* nodes and the relation-to-concept and concept-to-relation arcs and replaces these with a richly-typed set of n-ary arcs, all of which are a subtypes of the prototypical relation shown in Figure 39. The advantage of this interpretation is that Constraint Graphs can use the intrinsic constraints on arcs to model semantic typing of concepts and relations, restricting the arguments of relations (arcs) to conform to the type hierarchy (as defined by the is-a arcs). Although this semantic extension *could* be achieved using the first (traditional) interpretation, it would have to be done with parameterized ad-hoc constraints which would be arguably "unnatural" to Constraint Graphs' normal paradigm. It would also be comparatively

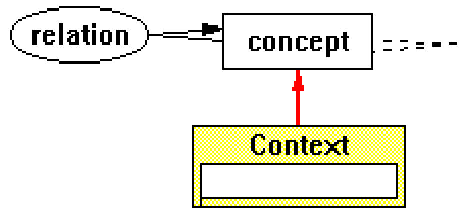
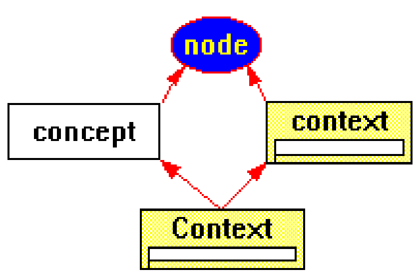


Figure 39: The Constraint Graphs top-level ontology for Conceptual Graphs

complex for the end user to specify. Therefore, the alternate, relations-as-arcs interpretation is judged preferable for Constraint Graphs and this is the one that is discussed throughout the remainder of this section.

In the relations-as-arcs model, there are only four primary types of components in Conceptual Graphs: *concepts, relations, coreference arcs,* and *contexts*. Figure 39 is a Constraint Graphs definition for these four component types. The unusual features (as compared to the gIBIS example) are:

- The *relation* component of Figure 39 is an arc, not a node. *Relation* is just a labeled arc where the label has a surround shape. The *relation* arc starts at *concept* and ends at *concept*.
- The *coreference* arc is the unlabeled dotted line that appears on the right-hand side of the *concept* node. As with *relation*, it both starts and ends on the *concept* node.
- *Context* is a Constraint Graphs context that inherits from the *concept* node. This is perfectly acceptable since concept<node, context<node, Context<context, and Context<concept forms a consistent type lattice:



The important attributes of the Conceptual Graphs definition are given in Table 4. Since the definition of Conceptual Graphs dictates that all concepts have rectangular surrounds, the shape attribute is set as constant for *concept*. This prevents sub-type concepts from overriding the attribute and changing the shape: to do so would interfere with the readability of the map. Likewise, other shape and connector attributes are also tagged as constant.

Table 4 also mentions one constraint. ParentOfIndividual is applied to the coreference arc because the coreference arc cannot be specialized in the Conceptual Graphs framework.

In Sowa's Conceptual Graphs, the ontology (or type lattice) is not included. Instead, the Conceptual Graphs are assumed to be accompanied by a separate semantic net that specifies the type lattice. In contrast, the Constraint Graphs version smoothly integrates these two separate graph structures into a single graphing space. This is very natural for the Constraint Graphs structure. So, a Constraint Graphs implementation of Conceptual Graphs might be considered an extension of Conceptual Graphs' graphical notation (but

Table 4: The attributes of the Constraint Graphs top-level ontology for Conceptual Graphs

Name	Type	Level	Attributes	Value	Flags
concept	node	3	shape	rectangle	const
relation	arc	3	shape	ellipse	const
			connector	directed labeled	const
coreference	arc	3	connector	plain	const
			constraints	ParentOfIndividual	
context	context	3	shape	frame	const

not an extension to Conceptual Graphs theory, since Constraint Graphs does not implement the whole theory – just the syntax and some of the semantics).

In modeling Conceptual Graphs, the "standard" level designations need to be extended. In the "standard" designations, there are two human roles: visual language designer and end user. In Conceptual Graphs, it makes sense to divide the end user role further into a user responsible for the semantic net, and a more tradition Conceptual Graphs user who is only responsible for the domain. Thus, there are three roles:

- visual language designer (responsible for the Conceptual Graphs definition, Figure 39)

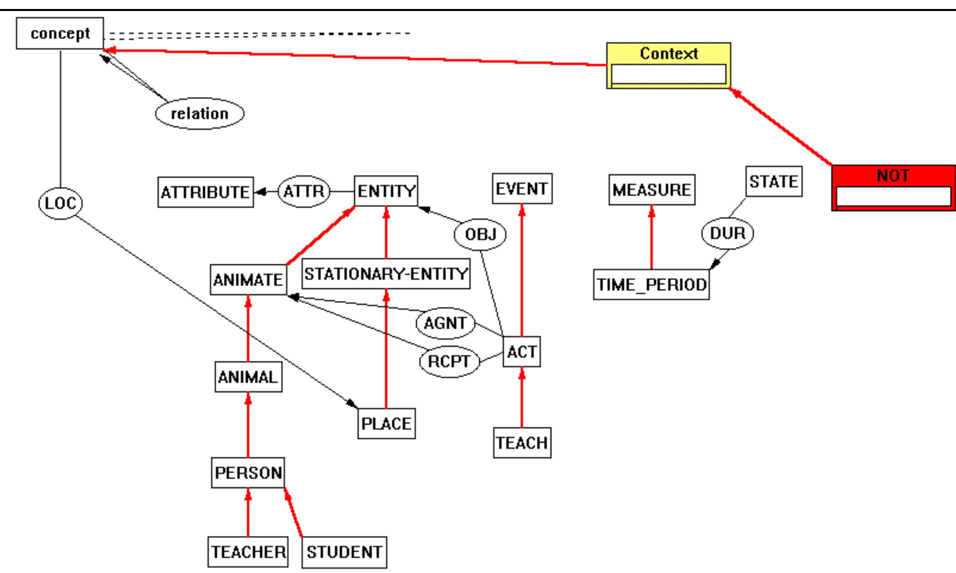


Figure 40: A Constraint Graphs schema for Sowa's Conceptual Graphs and some of top-level type hierarchy from Sowa (1984)

Table 5: Suggested level settings for Conceptual Graphs

user type	new menu		default level	locked
	min	max		
visual language designer	1	3	3	1
ontology designer	3	4	4	3
end user	4	4	6	4

- ontology designer (responsible for the domain ontology or semantic net, Figure 40)
- end user (responsible for using the ontology to construct meaningful expressions, Figure 41)

Table 5 gives the suggested level settings for each of the roles.

Figure 41 shows an example of a Conceptual Graph as represented by the Constraint Graphs program. The graph expresses the sentence "No student read the book the teacher wrote." The Constraint Graphs ontology used is shown in Figure 40. This ontology is taken from Appendix B of Sowa's 1984 book (Sowa 1984). The user who constructed Figure 41 had to add the new types WRITE<ACT, READ<ACT, and BOOK<ENTITY.

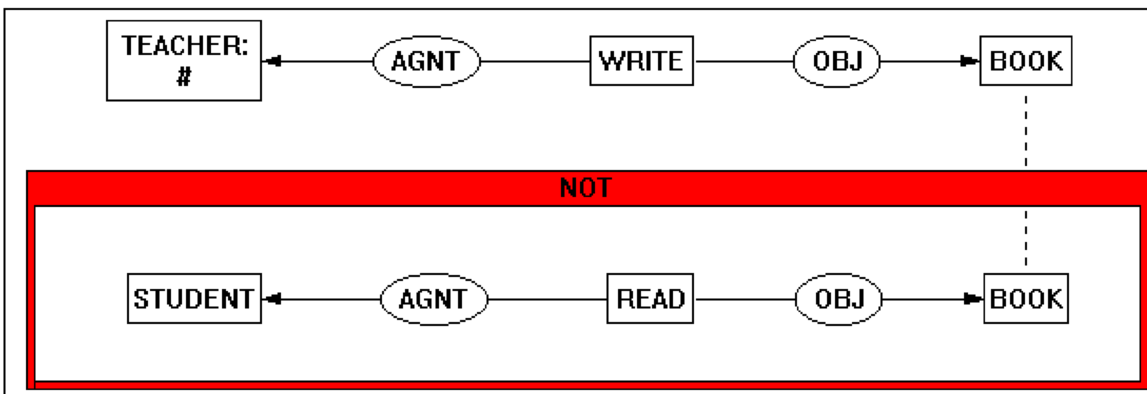
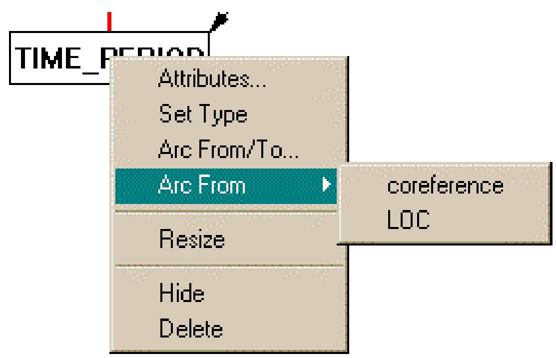


Figure 41: An example of a Conceptual Graph using Constraint Graphs

The ontology of Figure 40 is very straightforward. To avoid clutter, the type hierarchy omits the *isa* arc between concept and relation and their immediate subtypes (all of the rectangles and ellipses that aren't at the "from" end of an *isa* arc).

The Constraint Graphs program is capable of informing the user of all legal links between arbitrary nodes. A little experimentation with this feature quickly points out a few problems with the ontology which appears in the appendix of Sowa's 1984 book. For example, Sowa defines the LOC (location) relation as `[T]->(LOC)->[PLACE]` as it is in Figure 40 (T is *concept* in the figure). Constraint Graphs can infer all the possible arcs that a particular component may participate in. If the user requests this information for TIME_PERIOD, Constraint Graphs will infer TIME_PERIOD may be at the "from" side of either *coreferent* or LOC:



Clearly, LOC is not appropriate since a TIME_PERIOD should not have a location. This same problem applies to ATTRIBUTE, MEASURE, STATE, and all of their subtypes. It would make good sense to introduce a common supertype for ENTITY and EVENT to anchor the LOC relationship to, instead of anchoring it to T (*concept*). For example CYC (Cycorp 1996) defines "SpatialThing" as the supertype of "PhysicalEvent" and "PartiallyTangible" (which roughly correspond to Sowa's EVENT and ENTITY, respectively).

The implementation of Conceptual Graphs covers only the fundamental syntax and some of the semantics (type lattice constraints) of the Conceptual Graphs formalism. This level of support presented no particular difficulty to the Constraint Graphs framework, once it was recognized that relations could be modeled as arcs instead of nodes (as described at the beginning of this section). However, since much of the syntax of Conceptual Graphs is captured in node labels, a deeper modeling of Conceptual Graphs would involve several ad-hoc constraints to constrain the syntax of node labels (which has turned out to be not particularly difficult either – see Section 7.3.4). Furthermore, label syntax can interact with graphical syntax in the Conceptual Graphs implementation: an individual marker in a concept node label should preclude that node being used as a type (as the destination end of an *isa* arc). Even this can be handled using ad-hoc constraints, although not particularly elegantly. A more elegant method of constraining label syntax, and allowing label syntax to interact with graphical syntax, would be desirable; though it is not clear that such a method exists.

It may be feasible to model the full semantics of Conceptual Graphs using only minimal extensions to Constraint Graphs. This is described in the "future work" section (Section 9.3.3).

7.3 KDraw

KDraw (Gaines 1991b) is an interactive, visual, term subsumption language. It is based on the (textual) CLASSIC language (Borgida et al. 1989). KDraw has 6 main types of nodes, and eleven relationship types as depicted in Figure 42. All of the KDraw types (below the heavy line in Figure 42) belong to level 3. The fundamental KDraw node types are *concept, primitive, individual, constraint, role,* and *rule*. KDraw has eleven relations. The termination rules for these are shown in Table 6.

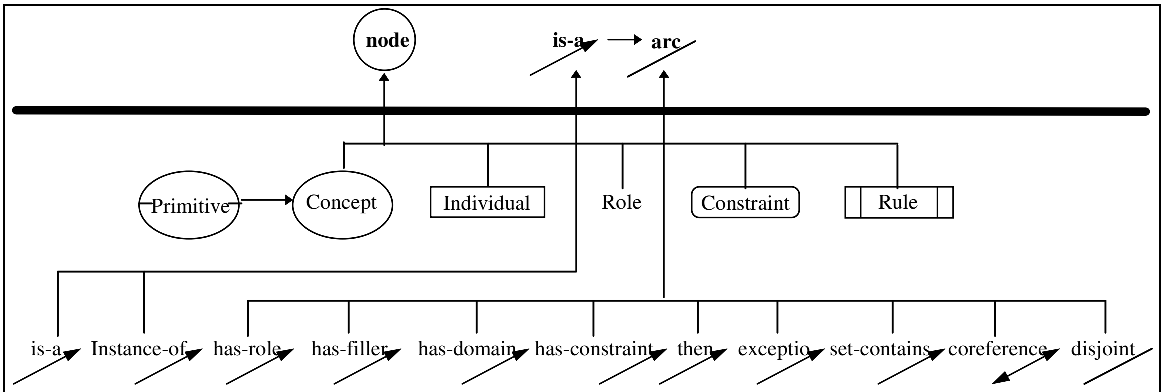


Figure 42: A Constraint Graphs model of KDraw. The vertical and horizontal arcs represent the subtype relation. Objects above the heavy line are the types of the Contraint Graphs domain; objects below the heavy line are system types in the KDraw domain.

Two of the relations are particularly interesting: this is the first example of subtypes of the *isa* arc being used in language definition. The considerable semantics (rules) associated with the *isa* arc (*isa* arcs cannot form cycles, *isa* arcs are binary, etc.) are automatically inherited by the subtype *isa* and *instance-of* arcs.

Figure 43 shows an actual Constraint Graphs schema for the component types in KDraw. Node labels all begin with upper-case letters; arcs (relationships) are pictured as all lower-case letters. The surrounds (shapes) of nodes are similar in shape and style to those used in KDraw. Note that although *role* has no surround, it is a node and should not be confused with an arc. Constraint Graphs allows nodes with no surrounds (as in the present case) and arcs with surrounds (as in the Conceptual Graphs example) – it's up to the visual language designer to avoid the potential confusion that might arise.

Table 6: Legal terminators for the KDraw arcs

Arc Type	From Node Type	To Node Type
isa	Concept	Concept
instance-of	Individual	Concept
has-role	Concept, Individual, Rule	Role
has-filler	Role	Individual
has-domain	Role	Concept
has-constraint	Role	Constraint
if-then	Concept	Rule
exception	Rule	Rule
set-contains	Set-Constraint	Individual
coreference	Concept	Concept
disjoint	Concept	Concept

Arcs in KDraw are all unlabeled (they are only labeled here for clarity), and for most part, are binary directed arcs (arrows) unambiguously distinguished by the (visual) type of their terminal objects. The cases where arcs types are not unambiguous are those in which the arc types have both terminals at concepts: *isa*, *disjoint*, and *coreferent*. These are visually disambiguated by *isa* being a directed arc, *disjoint* being an undirected arc, and *coreferent* being a bidirected arc.

Of course, each of the objects shown in Figure 42 must be constrained to conform to the graphical syntax of KDraw. This is done in the Constraint Graph schema of Figure 43. For nodes, this is trivial, for the arcs alone suffice to control the syntax. (Any nodes could stand alone and unattached without violating any KDraw rules, though they wouldn't be very meaningful.) The only predicates applied to nodes are regular expression constraints, which apply to the labels the end user puts on the various *Constraint* nodes. For the most part, arcs can be constrained by merely connecting their terminals to the appropriate type object. These terminals are (almost) precisely those shown in Table 6. Complications in the actual definition (Figure 43) arise due to the need to introduce additional types:

- to encapsulate the "polymorphic" *has-role* relationship,
- to handle the natural inheritance of visual attributes from *Concepts* to *Individuals* through *instance-of* arcs without giving *Individuals* all the semantics of *Concepts* (to be detailed below),
- to avoid cycles in the *isa* hierarchy when defining the *isa* KDraw relation, and

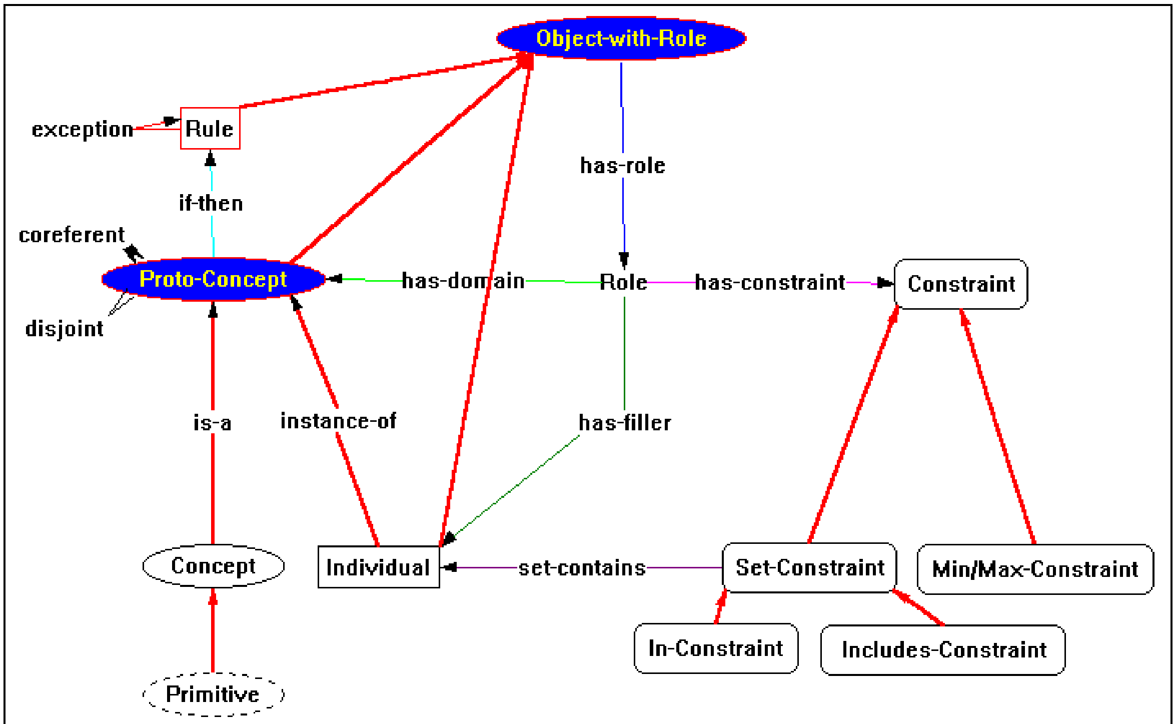


Figure 43: A Constraint Graphs schema for KDraw

- to limit the syntax of labels on KDraw *constraint* nodes.

The next four sub-sections describe these issues in more detail.

7.3.1 "Polymorphic" Relations

Several different, otherwise unrelated, node types may appear on the "from" side of an arc connecting to a *Role* node type. A *Rule, Concept, Primitive,* and *Individual* may all connect to a *Role*. (But we can ignore *Primitive* because it is just a subtype of *Concept*.) There is no way to draw the prototype (defining) *has-role* arc with three tails and one head without distorting its meaning (its arity is meant to be two, not four). One way to get around the problem is to define three different *has-role* arcs: *has-role-rule, has-role-concept,* and *has-role-individual*. But this is not very elegant and seems to be contrary to the "natural" way of thinking about this relation.

A more natural solution is to encapsulate the commonality between rules, concepts, and individuals: they are all "things with roles". One can therefore define a supertype of all three called *Objects-with-role*. This is exactly what's done in Figure 43. The new type *Object-with-role* functions as the "from" terminal for the *has-role* arc, and *Rule, Concept, Primitive,* and *Individual* inherit the ability to have a role from *Object-with-role*.

7.3.2 Natural Attribute Inheritance

A second problem has to do with natural inheritance. Note that, in Figure 43, *Individual* is not a subtype of *Concept*, but if the *Individual George* is an instance of *Child*, it might be reasonable for the user to expect *George* to inherit the attributes of *Child*. In particular, one might expect that the visual attributes will be inherited: If one shades *Person* by giving it a *fillColor* attribute, then one expects the attribute to be inherited by *Child* through the *isa* arc and then to be inherited by *George* thought the *instance-of* arc, as it is in Figure 43.

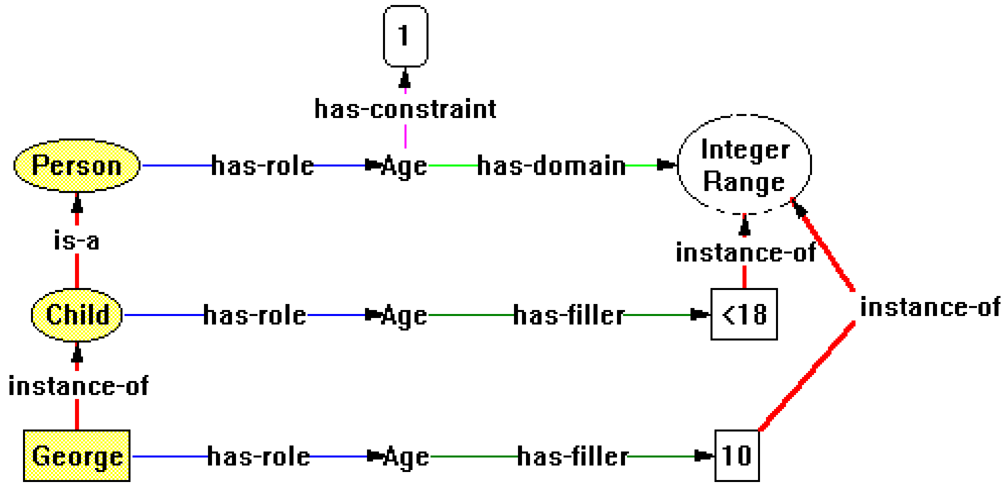


Figure 44: An example of KDraw using Constraint Graphs

Naively, one may consider making *instance-of* a simple subtype of *isa*, but this will not work. If *instance-of* were a simple subtype of *isa*, then *George,* in Figure 44, would inherit the ability, for instance, to be associated with a *if-then* arc, and this is not allowed (concepts have rules, individuals do not have rules). Thus, when the user shows the "Arc from" menu, the choices would be *disjoint, coreferent, has-role, if-then, isa,* and *instance-of*. But the only choices should be *has-role* and *instance-of*.

On the other hand, *not* making *instance-of* a subtype of *isa* means that individuals will not inherit visual attributes from the concepts that they are instances of. It seems there is no solution to the dilemma.

The problem lies in the modeling. Logically, there are *two* type lattices involved, not just one. One type lattice is used to determine arc termination eligibility, and is called the KR (Knowledge Representation) ontology. The other is used in attribute inheritance, and is called the domain ontology. The two ontologies are completely coincident except for the *instance-of* arc, which is in the domain ontology, but not the KR ontology. In the software, this scheme is supported by *ontology filters* as described in Section 5.3.

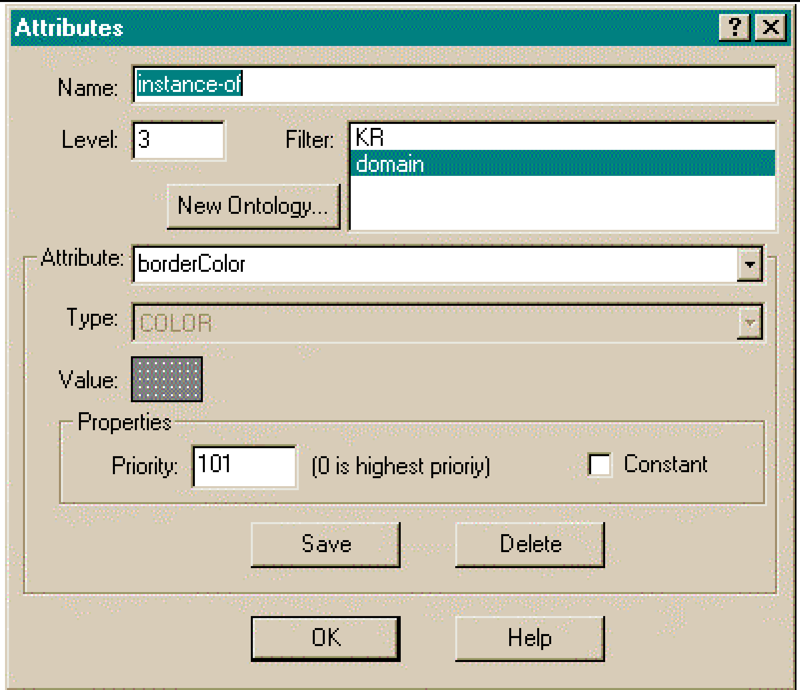


Figure 45: The attributes dialog for KDraw's *instance-of* arc

In practice, all that need be done is tag the prototype *instance-of* arc with the *domain* ontology filter as it is in Figure 45 where only the domain ontology is highlighted in the filter box. By default, all *isa* arcs are tagged with the universal ontology filter (where *all* ontologies are highlighted). This effectively solves the problem: attribute inheritance will occur though *instance-of* arcs because they are in the domain ontology, but *instance-of* arcs will not be considered when the system is performing type inferences, such as in arc termination eligibility. Using ontology filters neatly solves the problem without undue work on the user's part because different type lattices are normally highly coincident.

By the way, in Figure 43, the *instance-of* arc is drawn from *Individual* to *Proto-Concept,* and not to *Concept*, because *Concept* has its *shape* attribute tagged as constant. If *Individual* were to inherit this shape attribute from *Concept*, it could not override the shape to be its distinct rectangular shape. Therefore, the prototype *instance-of* arc must be drawn between *Individual* and some supertype of *Concept* that does not have a constant shape – *Proto-Concept*.

7.3.3 Avoiding *isa* Cycles

A KDraw *isa* arc may be drawn between any two concepts. Naively, one would just draw the prototype KDraw *isa* from the prototype *Concept*, back to the prototype *Concept*. But this is a cycle, and cycles are not allowed in type lattices. Fortunately, there's always a simple way out of the kind of dilemma: Just create a supertype of the target component, and connect the "circular" arc between the target and its supertype. This strategy has been used in Figure 43 to solve the problem of cycles.

7.3.4 Limiting the Syntax of Labels

Unlike most node labels in KDraw, constraint labels must conform to a very specific syntax. There are three types of constraints: cardinality constraints, subset ("in") constraints, and superset ("includes") constraints. Each of these types requires a specific syntax of their node labels. This is easily accommodated by the ad-hoc constraint "NameRegularExpression", which is a parameterized function object in the Constraint Graph's standard constraint library.

To attach the ad-hoc constraint, "NameRegularExpression", to a component, the user brings up the component's attribute dialog box (Figure 45) and selects the attribute "constraints". The "value" part of the dialog changes to an edit button, which the user can press to bring up a constraint editor. Figure 46 shows the constraint dialog for KDraw's *Min/Max-Constraint* (cardinality) node. To edit parameters of the "NameRegularExpression" constraint, the user chooses "NameRegularExpression" from the left-hand panel and then presses the "Edit Constraint" button. This brings up the NameRegularExpression parameter dialog box shown in Figure 47.

Ad-hoc constraint parameter dialog boxes are custom to each constraint. Figure 47 shows the one for "NameRegularExpression". It is just a multi-line edit box where the user may enter any number of regular expressions, one per line. The constraint merely checks to see that the component's name conforms to at least one of these regular expressions. Figure 47 shows the regular expressions needed to describe a KDraw *Min/Max-Constraint*. The first two expressions are required to accommodate the default names that might be assigned to the node upon creation. Including these two expressions allows the user to create the node without immediately specifying the actual constraint. The remaining three lines allow the user to express "at most *n*", "at least *n*", and "exactly *n*" cardinalities, respectively.

Of the three languages covered in this chapter, KDraw presents by far the greatest challenge to Constraint Graphs. Although KDraw was studied as a model for the initial theory and framework design, its actual implementation in the framework caused several iterations back to the theory and specification stages. For example, the "multiple

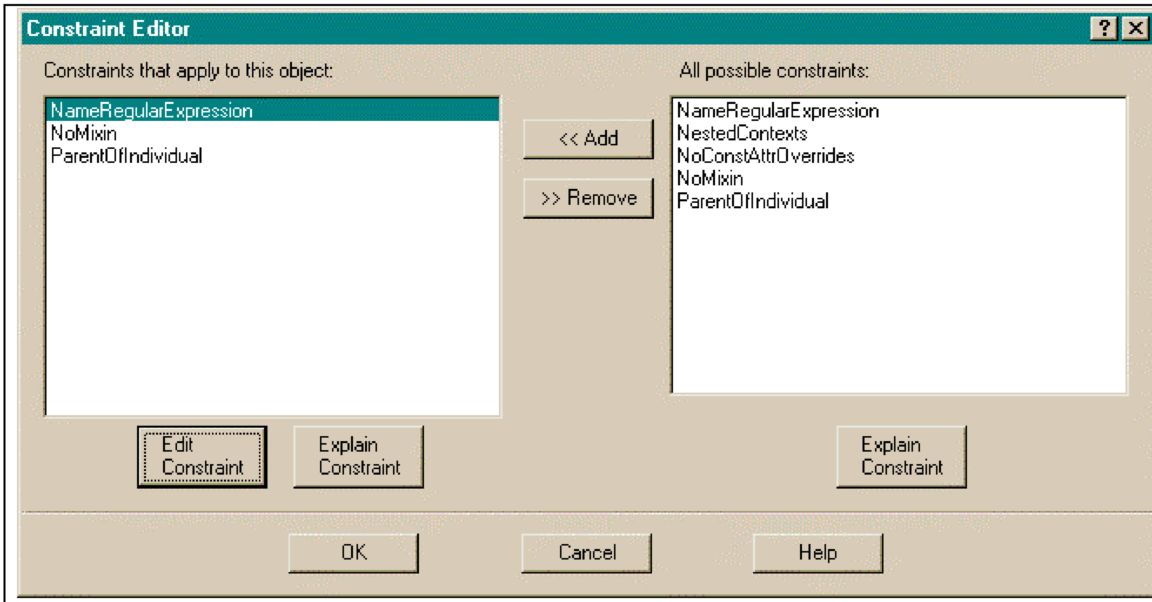


Figure 46: The constraint editor for the *Min/Max-Constraint* of the KDraw defintion

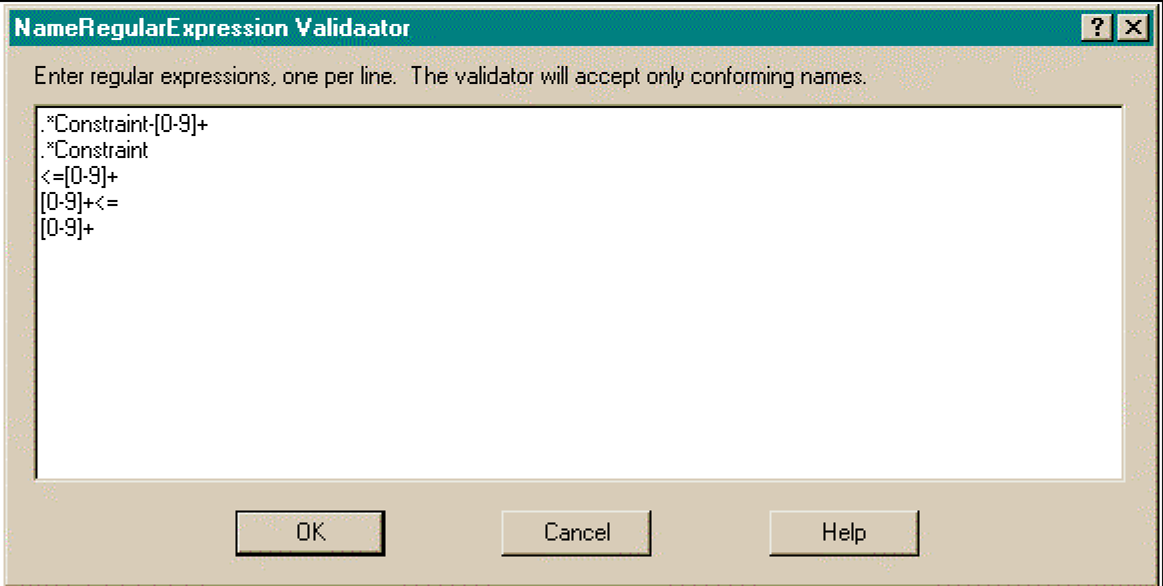


Figure 47: The "NameRegularExpression" constraint parameter dialog box for KDraw's *Min/Max-Constraint* constraint

ontologies" problem (Section 7.3.2) was unanticipated in the initial design and was only recognized when testing of the KDraw implementation yielded unexpected results.

7.4 Summary

This chapter has demonstrated the Constraint Graphs implementation of a variety of visual languages. These languages range from the simple to the complex, progressing though greater and greater challenges for Constraint Graphs. The example languages serve to demonstrate not only the features of the Constraint Graphs system, but also several strategies that can be used by visual language designers to implement languages in the Constraint Graphs system.

The first example, the gIBIS language for argument, is a very simple language. Its Constraint Graphs definition is described in detail, starting with the hidden, top-level abstractions and how they are used as the core to describe the concrete elements of the gIBIS language. The language syntax revolves around legal arc types, and these types are constructed to provide a usefully constrained user interface for the end user by specializing menus and specifying type inferences the system may make. Ad-hoc constraints are used to prevent unwanted subtyping. Constraint Graph's Levels are demonstrated by using them to hide details of the definition from the end user. Finally, the outcome of the gIBIS definition is explained by stepping through some of the operations that an end-user might perform in creating an actual gIBIS argument.

The second example, Sowa's Conceptual Graphs, is somewhat more complex than gIBIS. This example is used to describe Contexts, and to present a more complex example of the use of Levels. The Conceptual Graphs definition also serves to demonstrate more of the power of the inheritance relationships: inheritance is used by the end user to create a domain ontology. In contrast, the end-user in the gIBIS example merely unwittingly uses a discourse ontology created by the visual language designer.

The third example, KDraw, is a visual, executable, knowledge representation language, and is by far the most complex of the three languages. The KDraw definition demonstrates the use of abstract types to implement "polymorphic" relations, the use of ontology filters to create multiple type lattices in the same graph, the use of abstract types to avoid cycles in type lattices, and the use of ad-hoc constraints to limit the syntax of component labels.

The implementation of the three concept mapping languages in this chapter serves to both test and demonstrate the applicability of Constraint Graphs to implementing concept mapping languages. All of the current features of Constraint Graphs are exercised in the implementation of these languages. The value of implementing these languages in Constraint Graphs lies not only in the implementation per se, but also in the understanding of the target language that comes with the exercise of implementation itself. For example, testing with the implementation of Conceptual Graphs quickly pointed out some problems with Sowa's published ontology (Section 7.2) and the implementation of KDraw revealed an otherwise hidden structure of arc types. While the arc-centric structure of Constraint Graphs can be seen as unnatural with respect to the original conceptualizations of KDraw and Conceptual Graphs (which are both node-centric – see Section 9.3.2), the implementation in an arc-centric paradigm appears to be quite useful and insightful.

Chapter 8

Extensions and Variations

The previous chapter discussed the application of Constraint Graphs: languages that can be built on top of Constraint Graphs. This chapter aims at the use of these new languages and the environments in which Constraint Graphs may be used.

The first section discusses the Constraint Graphs extension that translates languages defined in Constraint Graphs to Z specifications. While the ability to translate into Z specifications is interesting, it is not necessarily particularly useful. But this exercise functions as a test case for translating visual languages into more traditional existent textual languages for further processing. For example, the KDraw or Conceptual Graphs language definitions described in the previous chapter could be translated into the native language of their respective engines. A KDraw domain could be translated into CLASSIC to create an expert system. A Conceptual Graphs domain could be processed by PEIRCE (Ellis & Levinson 1992), the Deakin Tool (Lukose 1991), or CGKEE (Munday et al. 1996).

The next two sections discuss two Constraint Graph variants. One variant is a Netscape browser plug-in that allows users to browse Constraint Graphs as active multi-media objects on the Web, modify them, and write them back to the server. The other variant is a multi-user version (which also runs on the Web) that allows multiple users to share the same Constraint Graphs workspace, interacting with the concept maps, and each other, in real time: each of them gets a constantly updated view of the map, reflecting the changes made by the others.

8.1 Translation to Z

The Constraint Graphs program is capable of capturing the *syntax* of other graphical formalisms by inferencing via the type lattice – but the reader should note that it is *not* intended to take on the deeper semantic nuances of any arbitrary graphical formalism. For example, although it can emulate the syntax of KDraw, Constraint Graphs cannot do the inferences of KDraw.

However, the Constraint Graphs program is relatively easy to extend to "compile" a graph into the native notation of a target formalism. As a case in point, it took only a single day to write an extension to output a concept map as a complete Z specification – this automatic generation produced the specifications found in Kremer (1996a; 1996b). The generated Z specification is an extension to the Constraint Graphs Z specification given in chapter 5.

The Z translator writes its output as an HTML file suitable for posting on the Web using the ZSL Z dialect (Jia 1995). The original Z language is full of esoteric mathematical symbols not found in many font sets, so Z is normally not very practical for viewing in non-specialized Web browsers. ZSL is an ingenious pure ASCII version of Z that substitutes very suggestive combinations of ordinary ASCII characters for all of the unusual mathematical symbols. Although it takes a little practice, it does not take a Z-literate person long to become familiar with ZSL. The convenience of ZSL being Web-friendly makes up for the slight loss in mathematical typographical elegance. Furthermore, ZSL is also an input language for Jia's ZTC Z type checker (Jia 1995), which means the Constraint Graphs Z translator output can be type checked by ZTC. An attribute that keeps the developer honest if nothing else.

By a fortuitous combination of syntactic conventions, ZTL can be embedded in HTML documents in such a way that Web browsers can display the document elegantly and ZTL can type check the ZTC code while ignoring the HTML code as comments. This is possible because ZTC ignores all lines that do *not* begin with a tab character. One only has to avoid using tab characters at the beginning of HTML code lines, and place tab characters at the beginning of all ZSL lines. HTML's <PRE> (preformatted) directive is used to enclose passages of (tab-prefixed) ZSL code, which maintains line-breaks in the ZSL code.

The Z translator's output starts by laying down HTML header information, then it includes Constraint Graphs' Z specification:

```
<html>
<head>
<title>A Z Specification for "KDraw4"</title>
</head><body>
<center><h1>KDraw4</h1></center>

This is a Z specification for the KDraw4 visual language definition.  It has
been automatically generated from a constraint graph by CONSTRAINT GRAPHS
(c)1996, Knowledge Science Institute, University of Calgary.  This specification
should typecheck using ZTC (c)1993-1995, Xiaoping Jia, DePaul University.

<pre>
        specification
<a href="graphsz.htm">
        include {graphsz.htm} %</a>
...
```

The Z translator then outputs a section describing the ad-hoc constraints, or *validators* that have been used in the language definition, but are otherwise not part of the Constraint Graphs definition. For example, the KDraw language definition (Section 7.3) uses the ParentOfIndividual constraint (but the Constraint Graphs specification does not), so ParentOfIndividual is listed in the validators section:

```
 ParentOfIndividual: VALIDATOR
--------------
 forall apply, owner: ARC_COMPONENT @ forall g: GRAPH0 @
   ParentOfIndividual(apply,owner,g) =
     if (apply /= owner => not (exists c:g.Contents @
          (apply->c in g.ancestorof0)))
        then PASS
        else FAIL
```

Z requires that all identifiers be declared, so the Z translator must make a second scan through the graph to pick up all new identifiers used in the concept map. There are two classes of identifiers: NAMES (used for components) and ATTRIBUTE_NAMES (used for attributes). Since ZTC is rather particular about identifier syntax (identifiers must begin with an alphabetic character and contain only alphabetic characters, numerals, and underscores) and Constraint Graphs is very liberal about identifier syntax (virtually anything is allowed), the Z translator must make up replacements for unacceptable names and create an identifier translation table. The identifiers section for Section 7.3's KDraw definition is:

```
| Concept, Primitive, Individual, Constraint, Rule,
| Role, mutex, coreferent, has_role, has_constraint,
| if_then, has_filler, has_domain, exception, In_Constraint,
| Includes_Constraint, Min_Max_Constraint, Set_Constraint,
| set_contains, Object_with_Role, instance_of, Proto_Concept : NAME

| borderColor, fillColor, shape, textColor, connector,
| lineColor : ATTRIBUTE_NAME
```

Finally, the Z translator outputs the schema that actually defines the target language. The following schema fragment is a fragment from the KDraw schema showing only the *Concept* and *Primitive* nodes and the *isa* arc. The complete specification is not shown because it's more than five pages long.

```
--KDraw4-----------------------------------------------------------
| FIRST_ORDER_TYPED_GRAPH;
| Concept: NODE_COMPONENT;
| Primitive: NODE_COMPONENT;
| is_a: ISA_COMPONENT;
| ...
|-----------------
| % Node "Concept" restrictions (id=0x8)
| Concept.Level = 3;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 0 @ % (0x0)
|       MakeAttribute(borderColor,x,101,{}) in Concept.Attributes;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 16777215 @ % (0xffffff)
|       MakeAttribute(fillColor,x,100,{}) in Concept.Attributes;
| % shape: seq char = "ellipse" @
| MakeAttribute(shape,attribute_value,101,{CONSTANT}) in
|                                     Concept.Attributes;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 0 @ % (0x0)
|       MakeAttribute(textColor,x,101,{}) in Concept.Attributes;
|
| % Node "Primitive" restrictions (id=0x9)
| Primitive.Level = 3;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 12632256 @ % (0xc0c0c0)
|       MakeAttribute(borderColor,x,101,{CONSTANT}) in
|                                     Primitive.Attributes;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 16777215 @ % (0xffffff)
|       MakeAttribute(fillColor,x,101,{}) in Primitive.Attributes;
| exists x:ATTRIBUTE_VALUE | pos_int~x = 0 @ % (0x0)
```

```
      MakeAttribute(textColor,x,101,{}) in Primitive.Attributes;

...

% Isa "is_a" restrictions (id=0x40)
is_a.Level = 3;
{ Upward, NoDanglingTerminals }
       subseteq is_a.Constraints;
GetTerminal(is_a,1) = Concept;
GetTerminal(is_a,2) = Proto_Concept;
MakeAttribute(connector,attribute_value,100,{}) in is_a.Attributes;

...

% the type hierarchy...
Concept parentof Primitive;
...
```

This schema first declares all the new components, then describes each component, and finally, describes the type hierarchy. Note that the type hierarchy does not seem to be complete – this is because a large part of it is implicitly declared in the other definitions.

The output concludes by adding standard HTML trailer text. The generated HTML file can be displayed as a standard Web document and can be type checked using the ZTC type checker.

Future work on this project includes adding an extension to translate a concept map into the (usually textual) native language of other systems, calling up those other systems for native processing and then translating the results back into a Constraint Graphs concept map. For example, KDraw concept maps could be translated into KDraw's underlying native notation (CLASSIC), a separate KDraw interpreter could be called (possibly elsewhere on the Internet), and the inferences could be written back into the original concept map. Section 9.3.4 expands on this concept.

8.2 Constraint Graphs as a Netscape plug-in

There can be little doubt that World Wide Web (WWW) browsers have caught on with unprecedented success as a popular means of accessing information on the Internet. It makes good sense to take advantage of this success, whatever cause one attributes to it, and use the Web as the base infrastructure for a repository of concept maps. Indeed, concept mapping tools should be cleanly integrated with all the other objects normally found on the Web: text, rich text, still images, animation, and HTML forms, to name just a few of the wide variety of multimedia objects that have become available on the Web.

8.2.1 Concept Maps Embedded in Netscape HTML Documents

The current versions of the Netscape WWW browser allows concept mapping programs, written in C++, to be cleanly integrated with the browser such that users can interact with the concept map as easily as they might interact with a clickable image (Netscape Communications Corporation 1997). Concept maps can be used just as though they were native objects to the browser program. Figure 48 shows a Constraint Graphs concept map embedded in an HTML document. This document and concept map describe part of a C++ class library. The concept map is actually an active instance of the Constraint Graphs program. Therefore, this concept map can be scrolled or re-arranged by simple direct manipulation commands.

All this is possible through the *plugin* Application Program Interface (API) provided by Netscape. Plugins are programs which register themselves with the Netscape browser in association with a specific MIME type (Borenstein & Freed 1993), and which are dynamically loaded by Netscape whenever an object of that MIME type is displayed. Plugins can be either embedded in an HTML document or displayed as a complete document themselves.

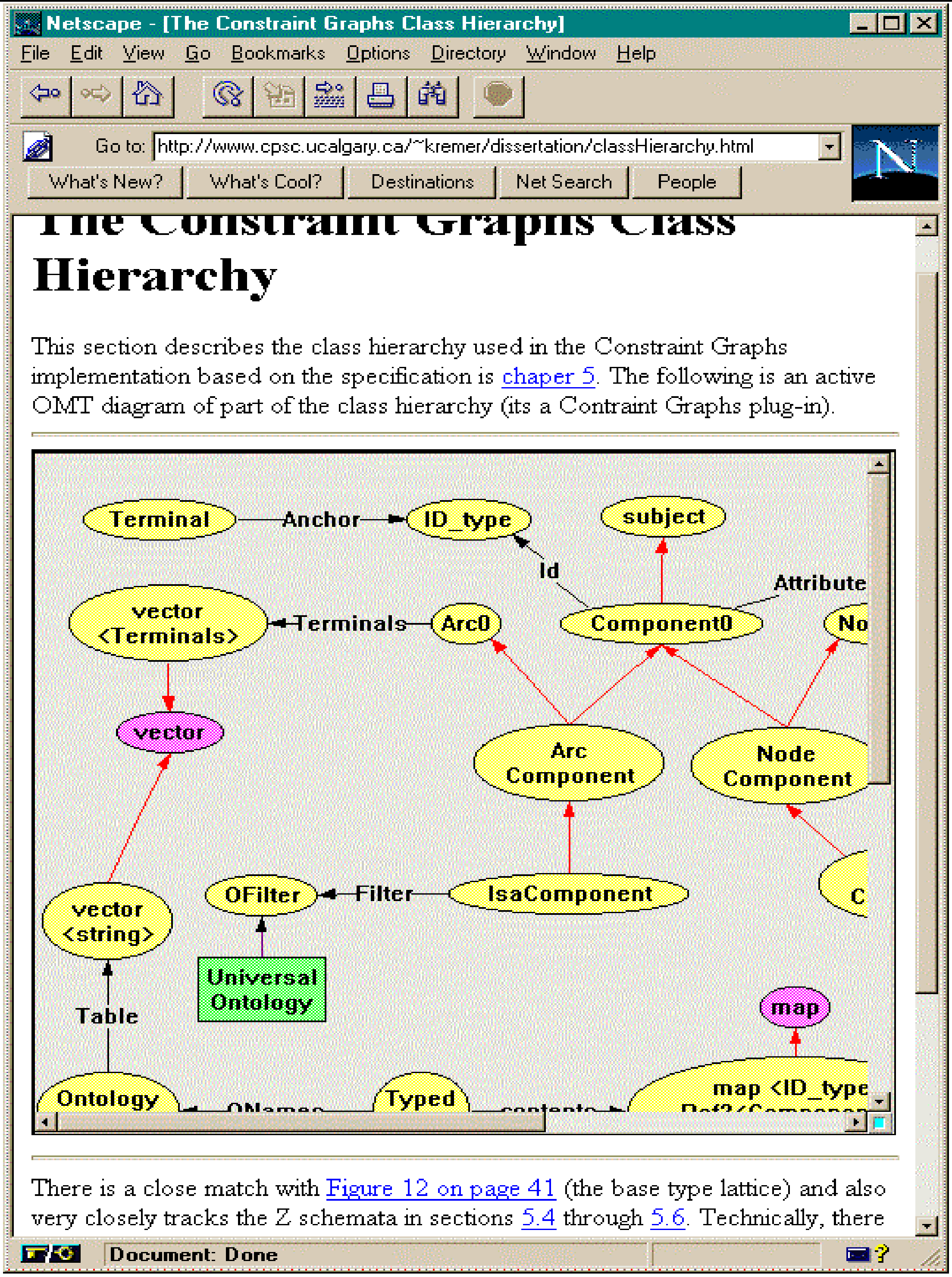


Figure 48: A concept map embedded in an HTML document describing part of a C++ class library.

With the version 3.0 release of the Netscape browser, plugins can masquerade as a Java (Sun Microsystems Inc. 1996) class. This facility is call *LiveConnect* (Netscape Communications Corporation 1996a). Java is an object-oriented programming language tailored for program migration on the Web, and is fast becoming the "native" Web language. In presenting itself as a Java class, a plugin program can make public any internal commands the programmer deems appropriate. Thus, a Java program can assume full control of a plugin program and provide specialized interfaces and operations to a concept map in a way that is convenient for Web programmers.

Furthermore, LiveConnect also provides open communication between Java and JavaScript (Netscape Communications Corporation 1996b). JavaScript is a scripting language that can be written directly into HTML documents, and is even easier to work with than Java. Since concept maps can masquerade as a Java class, and JavaScript can talk to Java, then JavaScript can exert control over concept maps. This allows the user to input elements of Web documents (primarily HTML forms (Grobe 1996)) to act as a large part of the interface to concept maps, giving the end user (or HTML script writer) an unprecedented degree of flexibility and ease of control.

Figure 49 shows a Constraint Graphs concept map embedded in an HTML document. In this example, most of the non-direct manipulation user interface elements that would normally appear as menus or hot keys have been transformed into commands available through a Java class. HTML forms input elements, such as buttons and selection boxes, are scripted with JavaScript to access these commands. For example, the button in Figure 49 labeled "save" is simply defined as:

```
<input type="button" name="save" value="save" onClick="document.map.save()">
```

which will call the save() method of the plug-in applet.

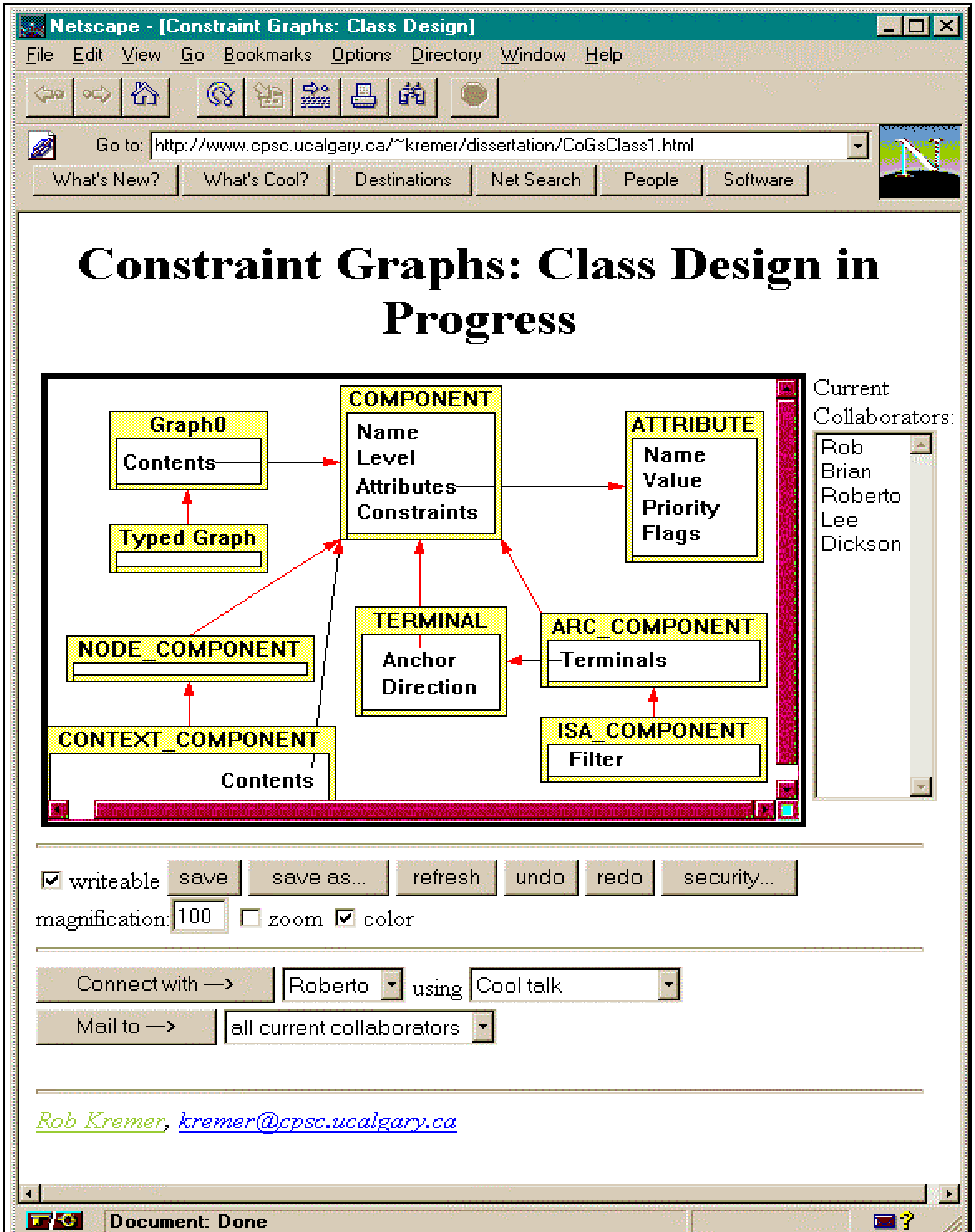


Figure 49: A concept map embedded in an HTML document using HTML forms input elements and JavaScript to achieve a custom user interface.

8.3 Groupware: multi-user, real-time interaction

While concept maps can be very useful as Web documents, their usefulness can be dramatically increased by making them multi-user interactive documents. That is, the traditional distinctions between document author and reader can be dissolved to create a *participant* role, where (sufficiently privileged) individuals can not only browse a concept map but also change the document and store the modified document back to the server. Furthermore, if two or more users happen to access a particular document at the same time, they should be able to smoothly move into a synchronous shared editing session.

8.3.1 Concept Maps Can Be Stored Back to the Server

Allowing users to update concept maps from a Web browser is accomplished through use of the CGI (Common Gateway Interface) available on most WWW servers. A shared concept map is not directly referenced by its normal URL; instead, a custom CGI program is referenced, and this reference is parameterized with a reference to the actual concept map document. When the URL is fetched by the client browser, it is the responsibility of the CGI program to return the concept map data on behalf of the server.

When the client wishes to save the concept map, the same indirect URL is used – but with the concept map data as part of the *post* message. The CGI program is called by the server, and the CGI program copies the data back into the original file on the server.

Of course, this simplistic approach is open to malicious or accidental disruption of the data. A host of security issues come to mind. While these issues have not been addressed so far in this research, there are many well-tested solutions elsewhere. Version control, which can be as simple as maintaining a new version of a file every time it is written back to the server, is one approach; public-key encrypted electronic signatures being verified on a write-back is another.

8.3.2 "Real time", Serendipitous Interaction

Merely allowing users to write concept maps back to the server is not enough. Besides the obvious dangers of users overwriting each others work, users need to be kept aware of other users concurrently accessing the concept map. Users may also wish to find out more about various aspects of the history of the map, such as who else has modified it.

Since shared concept map access is through a server CGI program, this program can take responsibility for recording access as part of the concept map's history, and for coordinating "real time" interaction between concurrent participants. Thus, clients can obtain information about past and current participants from the server (as in Figure 49). Furthermore, since the software architecture uses *command objects* (Gamma et al. 1995) to encapsulate all user updates to a concept map's state, these command objects can be sent to the server where they can be re-broadcast to all concurrent participants. In this way, each of the participants will receive all participants' commands, and all participants' concept maps will eventually arrive at the same state. (There are many finer points to this kind of interaction, but they will not be discussed here.)

This form of interaction has the advantage of being serendipitous; that is, users need not go to any special trouble to set up a "conference": instead, interactive sessions are automatically set up by virtue of concurrent access to the same concept map. (See Roseman & Greenberg (1996) for a discussion of this kind of interaction.)

8.4 Summary

The first section of this chapter discusses the Constraint Graphs extension that translates a Constraint Graphs concept map into a Z specification based on the Z specification for Constraint Graphs itself. This generated Z specification is an HTML document with embedded ZSL (a variant of Z) and can be displayed using any standard Web browser, and it can be used as input (in its entirety) to the ZTC type checker. The Z document serves as an example of simple translation from Constraint Graphs to the native language of other systems whose syntax is captured as a Constraint Graphs definition.

A variant of the Constraint Graphs program can be used on the Web. Constraint Graphs is embedded in an HTML document or is displayed *as* a Web document using the Netscape plugin API. This allows users to work with Constraint Graphs concept maps on the Web just as they might work with them on the stand-alone version of Constraint Graphs. Furthermore, Netscape's recent introduction of LiveConnect allows Constraint Graphs concept maps to interact with other elements, such as HTML forms objects, within an HTML document. This is accomplished by writing the Constraint Graphs plugin to emulate a Java class; as a Java object, the plugin can interact with both the Java environment and with JavaScript.

All of this is a useful concept mapping extension to the Web, but it still conforms to the Web's traditional author/reader roles. It is not difficult to further extend Constraint Graphs, using simple CGI programs to allow real-time, multi-user interaction where each of the users of a particular concept map may be updated with changes made by the other users in real time. This makes for a conference-like shared workspace environment, although a secondary voice channel is often necessary for useful interaction. An advantage of the scenario presented here is that, unlike many groupware tools, this is *serendipitous* interaction, where users do not have to spend undue effort setting up a "conference". Users need only agree to meet at a particular "place" (URL), or may happen to "meet" at a URL by chance.

Chapter 9

Conclusion

This chapter evaluates the Constraint Graphs framework and describes several potential areas of future work in the field. The evaluation addresses the objectives described in chapters 1 and 4, and describes how these objectives are met by the research work. The evaluation also discusses several issues related to the framework's efficiency, scalability, expressiveness, completeness, extendibility, and flexibility.

In addition to the evaluation, this chapter describes several avenues of future research based on the current work. The current theory should be challenged based on the implementation of other existing and newly designed concept mapping languages. There are also many possible alternative and variant theories that warrant investigation using variants of the software framework described in this thesis. It now seems apparent that the framework and theory could be easily and profitably extended to implement not just the syntax (Section 7.2), but all the semantics of Conceptual Graphs. Translators similar to the Z translator described in Section 7.3 could be used to link to other server programs to create systems that encompass both interactive syntax support (Constraint Graphs) and the "deep semantics" support of a language-specific (but non-graphical) server. A Java version of the CMap user interface could be used to increase Constraint Graphs portability to other environments. In addition, there are many avenues of research to pursue in order to make Constraint Graphs a powerful multi-user, collaborative tool.

9.1 Addressing the Objectives

The primary objective of this work was stated in chapter 1:

> "to provide the basis for concrete theories of concept mapping language syntax by developing a software framework for the development and implementation of concept mapping languages."

Two auxiliary objectives were derived from this: formal specification and disciplined development. The formal specification objective comes from the need for unambiguous description, since Constraint Graphs can support no theory if they are not unambiguously understood. The disciplined development objective comes from the need to have confidence in the correspondence between the specification and the program.

The requirements for the framework (which constitute a commitment to a minimal theory of concept mapping languages) were developed in chapter 4. From these requirements the specification was developed in chapter 5. While the specification does not cover the whole of the framework software, it does cover the essential part that implements the corresponding theory of concept maps. This includes the fundamental component types of a concept map: nodes, contexts, arcs, and the inheritance mechanism, *isa* arcs. The specification unambiguously documents:

- the type relationships among the primary component types (CONTEXT → NODE → COMPONENT ← ARC ← ISA)
- the intrinsic attributes of, and constraints on, each primary component type
- the method by which the primary components' type lattice can be extended to implement specific concept mapping languages
- the method by which the subtype/supertype relations is computed using *isa* arcs
- the constraints on which nodes of a particular type may appear on arc terminals (based on inheritance)
- the ad-hoc constraints and how they are added to components and computed through the type lattice

- the attribute/value pairs and how they are added to components and how access is computed through the type lattice
- the legality of components, and how it is calculated based on inheritance, arc terminal conditions, and ad-hoc constraints

This specification, given in chapter 5, fulfills the formal specification auxiliary objective.

Chapter 6 describes an implementation of the framework software. Although the software is written in a very different language (C++) from the specification (Z), the implementation is carefully structured to follow the specification as closely as possible. Unfortunately, following the Z specification's exact structure yields an implementation far too slow for practical purposes. This problem forced the implementation to cache the *isa* arc references, which was a slight divergence from the structure of the specification. Fortunately, taking only this shortcut was enough to increase the speed an average of two or three orders of magnitude (the computational complexity of the subtype calculation was reduced from fourth-order polynomial to almost linear). The vast majority of the implementation closely follows the specification structure. Both specification and implementation have been annotated with HTML anchor tags such that their trivial conversion to HTML documents allows two-way hyperlinks between the parts of the specification and the corresponding parts of the implementation.

However, there is an efficiency cost associated with maintaining the close correspondence between specification and implementation (see Section 6.5.3). Reducing this correspondence could significantly reduce the overall computational complexity, and may be necessary in the future as the framework is applied to larger data sets or is extended to different kinds of constraints.

In addition, chapter 6 describes the conceptual framework used in the design and implementation to make the program more understandable and high-level. Gamma's design patterns (Gamma et al. 1995) are high level organizational descriptions used in the design of the software. The C++ Standard Template Library (Nelson 1995; Stepanov & Lee 1995) is a very abstract and type-safe container library that was used in the implementation to avoid "reinventing the wheel" and offer a direct higher level of abstraction in the programming language.

The implementation has an (almost) direct correspondence to the specification. It also makes use of high-level conceptual frameworks in design and implementation. This disciplined approach yields an acceptable level of confidence that the implementation is a reasonable "concrete" facsimile of the specification.

Chapter 7 serves as a test of the primary objective of this work: to develop a software framework for the implementation of concept mapping languages. Chapter 7 describes, in detail, three concept mapping languages implemented in the Constraint Graphs framework. These languages vary in complexity from the simple gIBIS, a decision support language, to Conceptual Graphs and KDraw, both knowledge representation languages. The implementation of the syntax (and some of the semantics) of these three languages demonstrates Constraint Graphs' fulfillment of the primary objective of the thesis. This should not be taken as a claim that the theory will encompass all languages that one might want to include in the class of concept mapping languages: that will only come with further experience implementing more languages under the framework.

9.2 Issues

This section discusses the efficiency and scalability, expressiveness and completeness, extendibility, and flexibility of the current framework implementation. Most of the points within these areas are discussed elsewhere in this thesis; these are noted in the discussion.

9.2.1 Efficiency and Scalability

As has already been addressed in Section 9.1, efficiency is one of Constraint Graphs' concerns with respect to scaling up to large data sets. Efficiency has been purposefully sacrificed in favour of achieving a clean correspondence between the implementation code and the specification. However, the correspondence has been compromised because the early implementations proved much too slow. This forced references to *isa* arcs to be locally cached within component objects. In fact, the implementation maintains both the cached and uncached strategies by use of conditional compilation. The compilation is dependent on whether or not the macro symbol `FAST1` is defined, as seen in the following code excerpt of the declaration of the `Component0` class (the superclass of all component classes):

```
class Component0 : public Subject //inherit for the Observer/Subject pattern
  #ifdef FAST1
  ,public Observer
  #endif
  {
  friend class TypedGraph;
  public:
    typedef      set<Ref2<Attribute>, less<Attribute> > PolySetOfAttributes;
    typedef      PolySetOfAttributes::iterator AttrIterator;
    #ifdef FAST1
    typedef      list<IsaComponent*> Isas_type;
    typedef      Isas_type::iterator IsaIterator;
    #endif
    ...
protected:
    ID_type             Id;
    string              Name;
    unsigned int        Level;
    PolySetOfAttributes Attributes;
    #ifdef FAST1
    Isas_type           ParentIsas;
    Isas_type           ChildIsas;
    #endif
  };
```

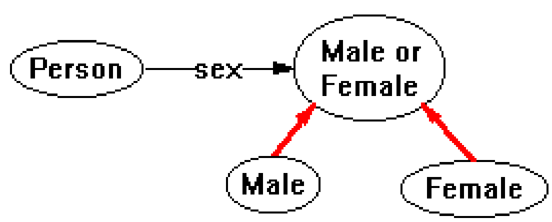
Here, the `Observer` class is a superclass of `Component0` only if this is a cached compilation. Likewise, the two lists of cached *isa* references and their relevant typedefs are only declared in a cached compilation. This measure allows for checking the cached version against the original uncached version during testing, and increases confidence in the equivalence of the specification and implementation. Section 6.5.3 discusses more technical details on cache optimization.

As Constraint Graphs is applied against larger and larger data sets in future work, it will undoubtedly show performance problems. The caching of other (non-*isa*) arc references are obvious candidates for efficiency improvements. Other efficiency strategies include:

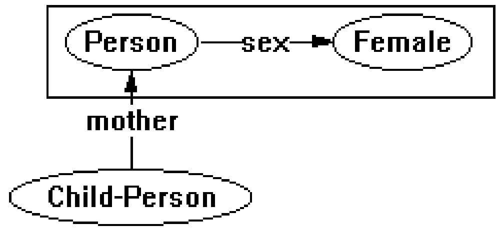
- The use of STL maps (red-black trees) and other time-efficient (but complex) structures instead of STL lists for such structures as attribute lists and graph component lists.
- The flattening and reordering of nested iteration structures that model the specification's nested universal and existential quantifiers.

9.2.2 Expressiveness and Completeness

The design of Constraint Graphs concentrates on arcs and which node types they may terminate on. A corollary of that is the attention to the type lattice and the subtype/supertype relationship, since typing is the basis of arc constraints. Complex arc constraints, based on more information than just the type of the arc and the type of the termination node, are not supported except by ad-hoc constraints. An example of a complex constraint might be the modeling of sex in type *Person* by attribute instead of by subtyping:



When modeled this way, the relationship *mother* is restricted to being from a *Person* to a *Person* whose *sex* relation terminates on *Female*, which could be drawn this way:



where the context box indicates the secondary restriction on the *Person* at the terminal of the *mother* relationship. Allowing this kind of specification can lead to arbitrarily complex structures, which leads to very high computational complexity in the constraint validation algorithm. Since this sort of specification appears to be fairly unusual (although it's hard to tell how much it would be used if it were readily available), it does not seem worth the additional complexity at this time – particularly since ad-hoc constraints can be used for the (apparently) rare cases where complex arc constraints are required.

Contexts have been specified as being hierarchical (nested) in Constraint Graphs. But there is no inherent reason why concept mapping languages need only hierarchical contexts: overlapping contexts are also possible (and more general). However, the author knows of no concept mapping language that allows overlapping contexts (few even allow contexts), but there are other graphical languages (Venn diagrams (Hammer 1995), for instance) that do allow overlapping contexts. While there is no current compelling reason to allow overlapping contexts, it may be necessary to roll back this commitment in the future. Fortunately, this would be easy: the nested contexts restriction is implemented as a simple ad-hoc constraint (Sections 5.6, 5.8, and 5.10), and could be removed or replaced by some other appropriate constraint. Even an end-user could remove the constraint, since it's a simple matter of removing it from the ContextComponents's "constraints" attribute: there is no programming involved.

Contexts are also problematic in another way: there is no obvious "right" way for context boxes to behave in the user interface. Currently, moving a context box moves all the components it contains as well as includes all components its post-operation geometry happens to contain. Resizing a context box includes (or excludes) all the components the post-operation geometry includes (or excludes). But there are compelling alternatives. Moving a context box *could* move all the contained components and "bump" all the components from its new location to maintain its containment state. Resizing a context box *could* compress all contained components or "bump" all excluded components to maintain its original containment state.

9.2.3 Extendibility

Besides the expected extendibility that comes with the object oriented programming paradigm, Constraint Graphs is easily extendible in at least three respects:

- Ad-hoc constraints
- Attribute/value pairs
- Library pattern

Ad-hoc constraints are implemented as instances of the abstract class *Validator*. Arbitrary constraints may be easily added to the system by sub-classing *Validator* and adding an instance of the new class to the global constraint library. (Since *Validator* is *clonable*, the system can produce exact-typed copies of a specific *Validator* subclass without knowing the specific type – see Section 6.5.2.) The system calls the *bool operator()* method to check for pass/fail status of the validator. The system does not have to worry about details of the validators: it merely keeps lists of validators and applies them as necessary in its validation algorithm (see Section 6.4.3). Thus, it is relatively easy to extend the system using validators with very minimal disruption to the program code.

Validators are stored in attribute/value pairs, as are all of the visual attributes (such as shape, fill color, and text color) used in the user interface. Attribute/value pairs are type-safe, identifier-indexable places where other programs or modules can easily store inheritable data within a Constraint Graphs concept map. This allows, for instance, the CMap user interface to store data (of which Constraint Graphs has no semantic knowledge) in a concept map; that data can be accessed later, or used concurrently by other views using the same concept map (see Section 6.1). Attributes and their values are also automatically inherited by subtypes of the component in which they are stored.

The library pattern, discussed in Section 6.5.2, allows for the easy extension of constraints and various graphical object types.

These three features, ad-hoc constraints, attribute/value pairs, and the library pattern, along with object-oriented design and implementation, combine to make the Constraint Graphs framework extendible and very flexible, as noted in the next section.

9.2.4 Flexibility

The points in the previous section on extendibility are also relevant to the issue of flexibility. In addition to those points, the Constraint Graphs design is flexible in its implementation of its intrinsic model. The model is very easy to modify because most of the component constraints, such as "acyclic *isa* arcs" and "strictly-nested contexts", are implemented as ad-hoc constraints, which are very easy to modify if the need arises. These constraints exist as part of the constraint library and are added to the base type objects (node, arc, *isa* arc, and context) during program initialization. This not only makes it trivially easy to remove or replace these constraints, but also makes it easy for other type objects to re-use the same constraints (without any programming).

9.3 Future Work

This section describes potential future work to be undertaken to extend Constraint Graphs beyond its present functionality.

9.3.1 Challenging the Theory

The theory of concept mapping languages presented in the Constraint Graphs theory invites challenge. The implementation of new languages under the framework acts as a concrete test of the theory's applicability to each new language. If a language cannot be implemented under the framework, the language does not fall under the domain of the theory. Either the language is not a concept mapping language, or it represents a failure of the theory and invites a revision or extension of it. Of course, the judgment of whether or not a particular visual language is rightfully a concept mapping language is subjective; but this is to be expected in the classification of complex phenomena such as visual languages.

Such challenges have played a major role in the development of the Constraint Graphs theory. For example, the theory was extended to include multiple ontologies when the implementation of KDraw showed the need for both a semantic and a visual ontology (as described in Section 4.3.6). The framework failed to allow the desired behaviour (*individual's* inheritance of attribute-value pairs from *concepts* while baring the application of *concept's* constraint behaviour). In this case, the failure indicated a deficiency in the theory, which lead to its extension to include multiple type lattices, which was implemented as ontology filters in the framework.

In addition to the languages described in Chapter 7, several other languages have been defined using the Constraint Graphs framework with varying degrees of success. One of these languages was a new language (which was originally designed using Constraint Graphs) called Annotated Flow Chart (Kremer, Lukose & Gaines 1997). This language is an extension to flow charts and presented no challenge at all for Constraint Graphs. In contrast, some of the object oriented notations were also implemented, and these presented some challenge to Constraint Graphs. The problem here involved contexts, which are probably underdeveloped in the present theory. In these implementations, methods where modeled as nodes that could only be contained inside classes or objects (which were modeled as contexts). While ad-hoc constraints could be used to solve the problem, it seems natural that there should be a way to model such constraints visually, just as arc termination constraints are visually specified in a natural way. Exactly how this could be done, or even *if* it should be done, is open to debate.

Contexts themselves present an issue in the theory. It is debatable whether or not contexts belong in a theory of concept maps. Many concept mapping languages lack the notion of context; for example, neither gIBIS nor KDraw contain any notion of contexts (although KDraw allows one to draw a context, it has no semantic meaning whatsoever). On the other hand, Conceptual Graphs embraces context as a fundamental notion. The current theory allows for the use of contexts, but contexts can be easily constrained away for languages that do not use context. The contexts in the theory follow Conceptual Graphs in that contexts are strictly nested and cannot be "overlapped" (Section 4.3.2.4). This is a contentious point since using this constraint does not admit such languages as Venn diagrams and Euler circles (Hammer 1995). However, elimination of these languages is probably justified in that it is easy to judge Venn diagrams and Euler circles as not being concept maps since their semantics (and lack of arcs) is substantially different from most "obvious" concept mapping languages.

There is much future work in implementation of both existing and new concept mapping languages in the framework. The expectation is that this will lead to a better and more complete theory of concept mapping languages.

9.3.2 Alternate Theories

There are many alternate possibilities in formulating a theory of concept mapping languages. Several alternatives have already been mentioned:

- *Omitting contexts from the theory*: This possibility does not admit Conceptual Graphs as a concept mapping language. (But Conceptual Graphs *should* probably be admitted.)

- *Relaxing the theory's constraint on hierarchical contexts (Section 4.3.2.4)*: This might admit languages like Venn diagrams and Euler circles, which are questionably classified as concept mapping languages.

- *Accommodating context constraints*: The present theory says nothing about constraining components to exist only in a particular context type or restricting a context type to contain only particular component types. While this can be done through ad-hoc constraints, supporting this ability through some "native" method (analogous to arc constraints) might be more appropriate.

There exist alternative theories that are far more fundamental than these. One of the more important possibilities is the theory implied by both KDraw and Conceptual Graphs: that nodes should "control" the graph syntax while arcs take a subservient role. In both these languages, arc types are largely neglected in favour of a rich set of node types. In KDraw, arcs are all binary and untyped and their meaning can only be inferred from the types of the nodes at their terminals (the exception is arcs between *concepts* which have three relations [*subtype*, *coreferent*, and *disjoint*] and are visual distinguished). In Conceptual Graphs, nodes take on the role of relations in a (almost) strictly bipartite graph and strictly binary arcs are untyped (except for the *coreferent* arc, which also breaks the bipartite nature of the graphs). Indeed, the arc-centric theory put forward in this thesis may be viewed as a challenge to the unstated node-centric theory implied by these and other concept mapping languages. Constraint Graphs takes the view that syntax (and some semantics) is constrained via arcs instead of nodes.

One may also formulate a theory that is the extreme opposition to the node-centric theory. It is possible to eliminate nodes from the theory entirely and view nodes only as arcs with an arity of zero. This arcs-only theory has the appeal of greater simplicity and homogeneity, since it eliminates one of the four fundamental types (nodes) in the theory. It also allows for a straight-forward description of arcs whose type label is an entire subgraphs, since such an arc is merely a context with arity greater than zero. The main drawback of the arcs-only theory is that it may be viewed as too radical and therefore difficult for many researchers to accept. This theory is viewed by the author as important, and will be tested by a variant of the framework in the near future.

9.3.3 Semantics of Conceptual Graphs

The description of the Constraint Graphs implementation of Conceptual Graphs in Section 7.2 models the syntax and some of the semantics (the type lattice) of Conceptual Graphs. In fact, of the four parts of a Conceptual Graphs canon (graphs, type lattice, conformity relations, and formation rules) Constraint Graphs very naturally models three of them. The graphs are easily expressed in Constraint Graphs and both the type lattice and conformity relations (*instance-of* relations) are modeled using Constraint Graphs' *is-a* arcs within the graphs themselves. The one part of the Conceptual Graphs canon that isn't modeled in Constraint Graphs is the formation rules. But, in fact, some of the formation rules *are* currently implemented in Constraint Graphs; other rules are easily and naturally modeled in simple implementation extensions to Constraint Graphs; and the remaining rules could be added as graph operations that might just as well be applied to other concept mapping languages.

It is therefore very compelling to add the remaining Conceptual Graphs formation rules as Constraint Graphs manipulation operators. This would achieve a complete implementation of Conceptual Graphs within Constraint Graphs. The four Conceptual Graphs formation rules and their likely implementation under Constraint Graphs is as follows:

- *Restrict*: The restrict operator is already implemented in Constraint Graphs since one of Constraint Graphs fundamental operations is subtyping by drawing an *is-a* arc from the subtype to the supertype. Any constraints violated due to the addition of such an *is-a* arc cause an immediate, run-time refusal of the operation.

- *Copy*: In the graphical environment of Constraint Graphs, the Conceptual Graphs copy operator naturally maps onto a standard copy-and-paste operation. While copy-and-paste has not yet been implemented in Constraint Graphs, it is a natural operation that would be implemented regardless of its need in a Conceptual Graphs implementation.

- *Simplify*: The Conceptual Graphs simplify operator could be implemented as an intrinsic function of the system: it would automatically be applied whenever the system detected two identically-typed arcs between two nodes. The implementation of such an intrinsic operator could be done trivially as a "noDuplicates" ad-hoc constraint on the Conceptual Graphs' top-level arc type, *relation*.

- *Join*: The join formation rule is the hardest to implement, but could be implemented either by an internal coreference arc or by merging two components into one. The first alternative would involve extending the theory to include the concept of coreference where the semantics of coreference arcs would be as conduits through which constraints are automatically propagated. The second implementation would merely involve a merge of two components into a single component with a single unique ID and changing all references in the graph to the new ID. The first operation seems slightly more appealing because it keeps more information about the history, and is therefore somewhat more "undoable" (without a command history list – both are equally "undoable" in the presence of a command history list).

One must cautiously avoid "polluting" the theory (and framework) with language-specific features, so a conservative evaluation of the generality of these new operators is in order. However, these operators all seem very generally applicable to most concept mapping languages (with the possible exception of *join*): *restrict* already exists; *copy* seems very natural in the graphical environment; and *simplify* can be implemented as a take-it-or-leave-it ad-hoc constraint. *Join* is the only one of the formation rules that is "active", in that it would involve the user actively issuing a new command to the system; nonetheless, it does appear to be a command that could be applicable to many other languages. As a case in point, it is semantically very similar to the KDraw *coreference* relation. It could be argued that the theory should be extended to encompass coreference as a fundamental concept, and this would subsume the join operator. Whether this should be graphically shown as a coreference arc or a merge of two components (or either) is more an issue of implementation and user interface than an issue of the theory.

9.3.4 Translators

A relatively simple translator has already been shown in Section 8.1. This translator converts a system developed in Constraint Graphs to a Z specification. While this may be useful for Z-literate users in understanding a system, the generated Z notation is considerably more verbose and hard-to-understand than the original Constraint Graphs definition. Although it is a useful demonstration, it is of limited utility.

More useful would be translators to other systems. For example, the KDraw Constraint Graphs system described in Section 7.3 could be translated into the KDraw version of CLASSIC, and passed to a KDraw engine where inferences could be made. If the translator could also translate back to native Constraint Graphs format, the inferences could be used as updates to the original graph, giving the user the impression that the graphical interface "knows" how to do inferences.

Figure 50 shows just such a situation. The Constraint Graphs process is using the KDraw definition described in Section 7.3, which describes KDraw syntax. The translator, built similarly to the Z translator, converts the user's definition into KDraw's CLASSIC-like representation language and sends the result to a KDraw process. The KDraw process is a server that may reside on the same machine, or may reside elsewhere on the Internet. A simple server protocol modeled on the HTTP protocol would suffice. The translator could also easily invoke the KDraw process (on the local machine) if it where not already running. This is not unlike the current KDraw "compilation" in the current version of KRS.

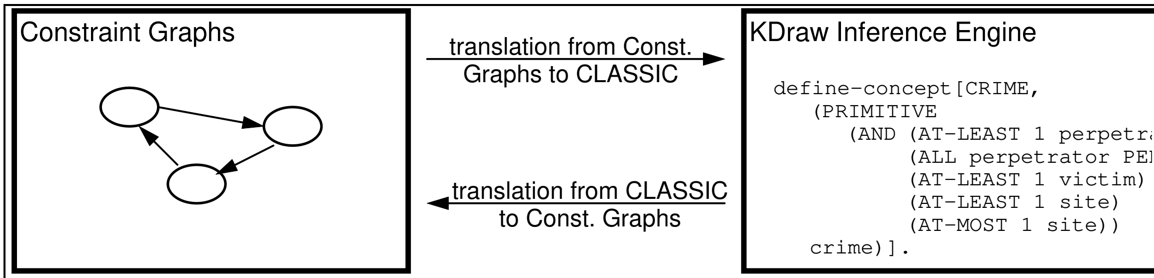


Figure 50: An example of Constraint Graphs as a KDraw front-end

9.3.5 Java Integration

Platform portability is not a large issue for the main part of Constraint Graphs since it is written is ANSII C++ and avoids the use of platform specific features. On the other hand, the CMap interface portion of Constraint Graphs is highly machine dependent because it cannot avoid using graphical routines and system routines (such as menu manipulators) that are highly platform-dependent. Steps have been taken to isolate this platform dependence: all system-dependent calls have been abstracted to platform-independent generalizations, and are contained in a code library and accessed indirectly by the programs through this library. This allows one to accomplish most of the work of a porting the framework to another platform by simply replacing the platform-dependent library.

Nonetheless, porting the interface to another platform still entails considerable effort both in terms of the initial work and ongoing support. Most of this work could be avoided by rewriting the interface in a platform-independent language, as Java is currently shaping up to be. As discussed in Section 8.2.1, re-writing all of Constraint Graphs does not seem to be a viable option; but, fortunately, Java is well equipped to interface with other languages, thanks to its "native method" facility. Java native methods are methods that do not have a Java body, but call a function in a run-time library using a standard protocol: they *act* just like Java methods, and so present a very clean interface to methods written in other languages.

It would not be difficult to interface a Java version of CMap to the current version of Constraint Graphs, where the two system communicate through "native" Java methods. In support of this goal, Java versions of the CMap library and the CMapper program have been developed at the University of Calgary (Flores-Mendez 1997). Since the Java programs include both a stand-alone version and a Web-browser embedded version, these programs form a good basis for Java-integrated Constraint Graphs versions.

9.3.6 Groupware

The Constraint Graphs groupware version described in Section 8.3 should not be viewed as anything more than a prototype. Much more work needs to be done to make Constraint Graphs a viable groupware application. Among the needed improvements are:

- Constraint Graphs' current undo command is appropriate only for single-user application. A more realistic multi-user undo protocol would not be difficult to implement, but it is still unclear which protocol is appropriate.

- The current synchronization model for the current multi-user Constraint Graphs version uses an optimistic protocol that incurs virtually no latency for the local user, but can incur very high overhead when more than a few users are involved. No dependency analysis is done when another user's command with a "past" time-stamp arrives; instead, the command list is unwrapped by undoing all the commands until the appropriate time for the new command is reached, and then the new command is executed, and all the "undone" commands are re-executed. In contrast, JMapper currently uses a pessimistic protocol that does not have the overhead problem, but can incur very high latency for the local user's commands on slow networks.

- Other-user awareness is an issue for all shared-workspace groupware systems. Users must be aware of other users in the workspace. This is particularly important for applications where there is no conference setup, and users may enter and leave the workspace asynchronously (like they can in the current Constraint Graphs model). This has been only superficially addressed, and considerable work must go into careful design of user awareness for a Constraint Graphs groupware version.

9.4 Summary

This thesis has described concept mapping languages as a sub-kind of visual languages. Even though concept maps are characterized by simple nodes and connecting arcs, the design space of concept mapping languages is large. This contention is supported by the large number of concept mapping languages in existence and the number of fields that concept mapping languages have been applied to. In order to design new concept mapping languages and to investigate variations on existing concept mapping languages, one must avoid the significant effort usually required to build graphical languages. This thesis addresses the problem by developing a general-purpose framework in which concept mapping languages can be quickly prototyped to a working application where the new (or modified) languages can be investigated.

The thesis has described visual languages, their value, semantic models, and classification. Concept mapping languages are distinguished from the more general visual languages by their distinct nodes and arcs. Several concept mapping languages are described in some detail. Since component type is an important concept in all formal concept mapping languages, the thesis describes type theory in the context of object oriented programming, graph grammars, and hypergraphs.

A detailed requirements analysis for the Constraint Graphs framework is developed in chapter 4. The formal specification (in Z) of Constraint Graphs is described in chapter 5, which amounts to a minimal theory of concept mapping languages. The design of the framework software is described in chapter 6. These chapters serve to describe the framework in detail.

Chapter 7 describes how the framework can be used by defining three different concept mapping languages: gIBIS, Conceptual Graphs, and KDraw. New languages can be defined by visually adding new nodes and arcs to the type lattice. These represent the component types of the defined language. Subtype/supertype relationships are visually defined by drawing *isa* arcs between the subtype component and its supertype component. These components will be used in type inference in the final language. A layering construct allows a language design to filter out generalization and other details that should no be available to the end user. More advanced language constructs can be defined using attribute/value pairs and ad-hoc constraints.

Chapter 8 describes extensions to Constraint Graphs, including translation of Constraint Graphs definitions into Z, a Netscape plug-in version of the framework, and the groupware version.

This final chapter has described how the research has met its original objective of creating a concept mapping framework by implementing several existing concept mapping languages. It also describes issues of efficiency, scalability, expressiveness, completeness, extendibility, and flexibility. Finally, some possible avenues of future work are described:

- challenges to the current theory based on the implementation of existing and newly designed concept mapping languages,
- alternate and variant theories,
- semantic extensions to the framework to accommodate the semantics of Conceptual Graphs,
- translators to (and from) the languages used by knowledge representation server programs,

- integration with a Java concept mapping interface, and
- advances in the groupware interface.

References

Abadi, M. & Cardelli, L. 1996. *A Theory of Objects*. New York, Springer.

Arnold, K. & Gosling, J. 1996. *The Java Programming Language*. Reading, Massachusetts, Addison-Wesley.

Aufaure-Portier, M. A. 1995. A High Level Interface Language for GIS. *Journal of Visual Languages and Computing* **2**(2): 167-182.

Axelrod, R. 1976. *Structure of Decision*. Princeton, New Jersey, Princeton University Press.

Banathy, B. H. 1991. Cognitive mapping of educational systems for future generations. *World Future* **30**(1): 5-17.

Barreiro, J., Fraley, R. & Musser, D. 1995. Hash Tables for the Standard Template Library. ftp://ftp.cs.rpi.edu/pub/stl/hashdoc.ps.Z, Rensselaer Polytechnic Institute and Hewlett Packard Laboratories, February, 1995.

Birtwistle, G., Dahl, O., Myhrhaug, B. & Nygaard, K. 1979. *Simula Begin*. Bromley, England, Chartwell-Bratt Ltd.

Booch, G. 1994. *Object Oriented Design With Applications*, Benjamin/Cummings.

Booch, G. & Rumbaugh, J. 1995. *Unified Method for Object-Oriented Development Set*, Rational Software Corporation.

Borenstein, N. & Freed, N. 1993. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. http://www.cis.ohio-state.edu/htbin/rfc/rfc1521.html, September, 1993.

Borgida, A., Brachman, R. J., McGuiness, D. L. & Resnick, L. A. 1989. CLASSIC: A Structural Data Model for Objects. Proceeding of 1989 SIGMOD Conference on the Management of Data. New York, ACM Press: 58-67.

Burnett, M. & Baker, M. 1994. A Classification System for Visual Programming Languages. *Journal of Visual Languages and Computing* **5**(3): 287-300.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. 1996. *A System of Patterns: Pattern-Oriented Software Architecture*. West Sussex, England, John Wiley & Sons.

Calcinelli, D. & Mainguenaud, M. 1994. Cigales, a Visual Query Language for a Geographical Information System: the User Interface. *Journal of Visual Languages and Computing* **5**(2): 113-132.

Cardelli, L. & Metchell, J. C. 1994. Operations on Records. Theoretical Aspects of Object Oriented Programming: Types, Semantics, and Language Design. C. A. Gunter & J. C. Mitchel. Cambridge, Mass, The MIT Press**:** 295-350.

Cardelli, L. & Wegner, P. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* **17**(4).

Church, A. 1956. *Introduction to Mathematical Logic*. Princeton, N. J., Princeton University Press.

Coad, P., North, D. & Mayfield, M. 1995. *Object Models: Strategies, Patterns, and Applications*, Yourdon Press.

Conklin, J. 1987. Hypertext: An Introduction and Survey. *Computer*(September, 1987).

Conklin, J. & Begeman, M. L. 1987. gIBIS: A Hypertext tool for Team Design Deliberation. *Hypertext'87***:** 247-251.

Constabile, M. F. & Catarci, T., Eds. 1995. *Special Issue on Visual Query Systems*. Journal of Visual Languages and Computing **6**(1).

Cook, W. R., HILL, W. L. & Canning, P. S. 1994. Inheritance Is Not Subtyping. Theoretical Aspects of Object Oriented Programming: Types, Semantics, and Language Design. C. A. Gunter & J. C. Mitchel. Cambridge, Mass, The MIT Press**:** 497-517.

Corrandini, A., Ehrig, H., Löwe, M., Montanari, U. & Padberg, J. 1994. The Category of Typed Graph Grammars and its Adjunctions with Categories of Derivations. Proceedings of 5th International Workshop on Graph Grammars and Their Application to Computer Science. J. Cuny, H. Ehrig & G. Engels. Williamsburg, VA, USA, Springer-Verlag**:** 56-74.

Courcelle, B. 1990. Graphs as Relational Structures: An Algebraic and Logical Approach. Proceedings of 4th International Workshop on Graph Grammars and Their Application to Computer Science. H. Ehrig, H.-J. Kreowski & G. Rozenberg. Breman, Germany, Springer-Verlag**:** 238-252.

Cox, B. J. 1991. *Object-Oriented Programming: An Evolutionary Approach*. Reading, Mass, Addison -Wesely.

Craig, I. D. 1991. *Formal Specification of Advanced AI Architectures*. New York, Ellis Horwood.

Craigen, D., Gerhart, S. & Ralston, T. 1993. *International Survey of Industrial Applications of Formal Methods*, U.S. National Institute of Standards and Technology.

Crimi, C., Guercio, A., Nota, G., Pacini, G., Tortora, G. & Tuccit, M. 1991. Relation Grammars and their Application to Multi-dimensional Languages. *Journal of Visual Languages and Computing* **2**(4): 333-346.

Cuny, J., Ehrig, H. & Engels, G. 1994. *Proceedings of 5th International Workshop on Graph Grammars and Their Application to Computer Science*. Williamsburg, VA, USA, Springer-Verlag.

Cycorp, I. 1996. Cyc Ontology Guide. http://www.cyc.com/cyc-2-1/intro-aaa.html, Cycorp Inc., August 7, 1996.

DeMarco, T. 1982. *Controlling Software Projects: Management, Measurement and Estimation*. Inglewood Cliffs, N. J., Yourdon Press.

Donini, F., Lenzerini, M., Nardi, D. & Nutt, W. 1995. *The Complexity of Concept Languages*, Deutsches Forschungszentrum fur Kunstleche Intelligenz GmbH.

Drewes, F. & Kreowski, H.-J. 1990. A Note on Hyperedge Replacement. Proceedings of 4th International Workshop on Graph Grammars and Their Application to Computer Science. H. Ehrig, H.-J. Kreowski & G. Rozenberg. Breman, Germany, Springer-Verlag**:** 238-252.

Eden, C., Jones, S. & Sims, D. 1979. *Thinking in Organizations*. London, Macmillan.

Eklund, P. W., Leane, J. & Nowak, C. 1993. GRIT: Toward a Standard GUI for Conceptual Structures. Second International Workshop on PEIRCE: A Conceptual Graphs Workbench. Laval University, Quebec, Canada.

Ellis, G. & Levinson, R., Eds. 1992. *Proceedings of the First International Workshop on PEIRCE: A Conceptual Graphs Workbench*. Las Cruccs, New Mexico.

Flores-Mendez, R. 1997. Programming Distributed Collaboration Interaction through the WWW (working title). MSc Thesis, Department of Computer Science, University of Calgary, Calgary, Canada.

Gaines, B. R. 1991a. Empirical Investigation of Knowledge Representation Servers: Design Issues and Applications Experience with KRS. AAAI Spring Symposium on Implementing Knowledge Representation and Reasoning Systems. Stanford, CA.

Gaines, B. R. 1991b. An Interactive Visual Language for Term Subsumption Languages. IJCAI-91. Sydney, Australia.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading Mass., Addison Wesley.

Genesereth, M. R. & Fikes, R. E. 1992. *Knowledge Interchange Formal, Version 3.0, Reference Manual*. Stanford, California, Computer Science Department, Stanford University.

Gerhart, S., Craigen, D. & Ralston, T. 1994. Experiences with Formal Methods in Critical Systems. *IEEE Software,* January 1994.

Goldberg, A. 1983. *Smalltalk-80 : the Language and its Implementation*. Reading, Mass., Addison-Wesley.

Graesser, A. C. & Clark, L. F. 1985. *Structures and Procedures of Implicit Knowledge*. New Jersey, Ablex.

Grobe, M. 1996. An Instantaneous Introduction to CGI Scripts and HTML Forms. http://kuhttp.cc.ukans.edu/info/forms/forms-intro.html, July 12, 1996.

Gronbaek, K., Hem, J. A., Madsen, O. L. & Sloth, L. 1994. Cooperative Hypermedia Systems: A Dexter-Based Architecture. *Communications of the ACM* **17**(2): 64-74.

Gronbaek, K. & Trigg, R. H. 1994. Design Issues for a Dexter-Based Hypermedia System. *Communications of the ACM* **17**(2): 41-49.

Gruber, T. 1995. Ontolingua Overview. http://www-ksl.stanford.edu/kst/ontolingua.html, Stanford KSL, December 15, 1995.

Gunter, C. A. & Mitchel, J. C., Eds. 1994. *Theoretical Aspects of Object Oriented Programming: Types, Semantics, and Language Design*. Foundations of Computing. Cambridge, Mass, The MIT Press.

Hailpern, B. & Nguyen, V. 1987. A Model for Object-Based Inheritance. Research Directions in Object-Oriented Programming. B. Shriver & P. Wegner. Cambridge, Massachusetts, MIT Press**:** 147-164.

Halasz, F. & Schwartz, M. 1990. The Dexter Hypertext Reference Model. Hypertext Standardization Workshop, Gaithersburg, MD.

Hammer, E. M. 1995. *Logic and Visual Information*. Stanford, California, CSLI Publications.

Hardman, S., Bulterman, D. C. A. & Rossum, G. v. 1994. The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model. *Communications of the ACM* **17**(2): 50-62.

Hart, J. A. 1977. Cognitive Maps of Three Latin American Policy Makers. *World Politics* **30**(1): 115-140.

Hayes, I., Ed. 1987. *Specification Case Studies*. Englewood Cliffs, N. J., Prentice-Hall.

Jia, X. 1995. ZTC: A type checker for Z (version 2.02). ftp://ise.cs.depaul.edu/pub/ZTC/, Telecommunication, and Information Systems, School of Computer Science, DePaul University, September 29,1996.

Kremer, R. 1993. A Concept Map Based Approach to the Shared Workspace. MSc. Thesis, Department of Computer Science, University of Calgary, Calgary, Canada.

Kremer, R. 1994. Concept Mapping: Informal to Formal. Third International Conference on Conceptual Structures, Knowledge Representation Workshop. University of Maryland.

Kremer, R. 1995. The Design of a Concept Mapping Environment for Knowledge Acquisition and Knowledge Representation. Banff Knowledge Acquisition Workshop. Banff, Alberta.

Kremer, R. 1996a. A Z Specification for KRS using Typed Graphs. http://www.cpsc.ucalgary.ca/~kremer/graphs/KRS_Z.html, September 30, 1996.

Kremer, R. 1996b. A Z Specification for the Conceptual Graphs based on of Typed Graphs. http://www.cpsc.ucalgary.ca/~kremer/graphs/CG_Z.html, September 30, 1996.

Kremer, R.. & Flores-Mendez, R. A. 1996. BNFs for KSI Map Storage Formats. http://ksi.cpsc.ucalgary.ca/local/software/storageBNF.html, Dec. 26, 1996.

Kremer, R. & Gaines, B. R. 1996. Embedded Interactive Concept Maps in Web Documents. Proceedings of WebNet'96: World Conference of The Web Society. San Francisco, CA, USA.

Kremer, R., Lukose, D. & Gaines, B. R. 1997. Knowledge Modeling using Annotated Flow Chart. International Conference on Conceptual Structures, Seattle, Washington, Springer.

Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B. & Nygaard, K. 1988. The BETA Programming Language. Research Directions in Object-Oriented Programming. B. Shriver & P. Wegner. Cambridge, Massachusetts, MIT Press**:** 7-48.

Lambiotte, J. G., Dansereau, D. F., Cross, D. R. & Reynolds, S. B. 1984. Multirelational Semantic Maps. *Educational Psychology Review* **1**(4): 331-367.

Laufer, K. 1995. A Framework for Higher-Order Functions in C++. UNEXIX Conference on Object-Oriented Technologies (COOTS). Monteray, CA.**:** 103-116.

Lieberman, H. 1986. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. Object Oriented Systems, Languages, and Applications, ACM.

Lukose, D. 1991. *Conceptual Graph Tutorial*. Geelong, Australia, Department of Computing and Mathematics, School of Sciences, Deakin University.

Lukose, D. 1993. Executable Conceptual Structures. Conceptual Graphs for Knowledge Representation. G. W. Mineau, B. Moulin & J. F. Sowa. Berlin, Germany, Springer- Verlag.

Lukose, D. 1995. Using Executable Conceptual Structures for Modelling Expertise. 9th Banff Knowledge Acquisition For Knowledge-Based Systems Workshop (KAW'95), Banff Conference Centre, Banff, Alberta, Canada.

Madsen, O. 1988. Block-Structure and Object-Oriented Languages. Research Directions in Object-Oriented Programming. B. Shriver & P. Wegner. Cambridge, Massachusetts, The MIT Press.

Marks, J. 1991. A Formal Specification Scheme for Network Diagram That Facilitates Automated Design. *Journal of Visual Languages and Computing* **2**(4): 395-414.

McNeese, M. D., Zaff, B. S., Peio, K. J., Snyder, D. E., Duncan, J. C. & McFarren, M. R. 1990. *An Advanced Knowledge and Design Acquisition Methodology for the Pilot's Associate*. Wright-Patterson Air Force Base, Ohio, Harry G. Armstrong Aerospace Medical Research Laboratory.

Meyer, B. 1992. *Eiffel: The Language*, Prentice Hall.

Munday, C., Cross, J., Daengdej, J. & Lukose, D. 1996. *CGKEE: Conceptual Graph Knowledge Engineering Environment User and System Manual*. Armidale, N.S.W., Australia, Department of Mathematics, Statistics, and Computing Science, University of New England.

Musem, M. A., Fagen, L. A. & Shortliffe, E. H. 1987. Graphical Specification of Procedural Knowledge for an Expert System. IEEE Computer Society Workshop on Visual Languages, Dallas, Texas, IEEE.

Myers, B. A. 1990. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing* **1**(1): 97-123.

Nelson, M. 1995. *C++ Programmer's Guide to the Standard Template Library*. Foster City, CA, IDG Books Worldwide Inc.

Netscape Communications Corporation. 1996a. Developer Information: LiveConnect. http://home.netscape.com/comprod/products/navigator/version_3.0/developer/mojava.html, September 28, 1996.

Netscape Communications Corporation. 1996b. Netscape JavaScript. http://home.netscape.com/comprod/products/navigator/version_2.0/script/index.html, September 28, 1996.

Netscape Communications Corporation. 1997. Extensions to HTML 3.0. http://home.netscape.com/assist/net_sites/html_extensions_3.html, March 10, 1997.

Nosek, J. T. & Roth, I. 1990. A Comparison of Formal Knowledge Representation Schemes as Communication Tools: Predicate Logic vs. Semantic Network. *International Journal of Man-Machine Studies* **33**: 227-239.

Novak, J. D. & Gowin, D. B. 1984. Learning How To Learn. New York, Cambridge University Press.

Petre, M. & Green, T. R. G. 1993. Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill. *Journal of Visual Languages and Computing* **4**(1): 55-70.

Pree, W. 1995. *Design Patterns for Object-Oriented Software Development*. Reading Mass, Addison-Wesley.

Price, B. A., Baecker, R. M. & Small, I. S. 1993. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing* **4**(3): 211-266.

Quillian, M. R. 1968. Semantic memory. Semantic Information Processing. M. Minsky. Cambridge, Massachusetts, MIT Press**:** 216-270.

Remy, D. 1994. Type Inference for Records in a Natural Extension of ML. Theoretical Aspects of Object Oriented Programming: Types, Semantics, and Language Design. C. A. Gunter & J. C. Mitchel. Cambridge, Mass, The MIT Press**:** 351-372.

Roberts, D. D. 1973. *The Existential Graphs of Charles S. Peirce*. The Hague, Mouton & Co. N. V.

Roseman, M. & Greenberg, S. 1996. TeamRooms: Network Places for Collaboration. Computer Supported Cooperative Work Conference. Boston, Mass.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J., Prentice-Hall.

Shneiderman, B. 1986. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* **16**: 57-69.

Smith, D. C. 1977. *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*. Stuttgart, Basel.

Snyder, A. 1988. Inheritance and the Development of Encapsulated Software Components. Research Directions in Object-Oriented Programming. B. Shriver & P. Wegner. Cambridge, Massachusetts, MIT Press**:** 165-188.

Sowa, J. F. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, Massachusetts, Addison-Wesley.

Sowa, J. F., Ed. 1991. *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. The Morgan Kaufmann Series in Representation and Reasoning. San Mateo, California, Morgan Kaufmann.

Spivey, J. M. 1989. *The Z Notation: A Reference Manual*, Prentice Hall.

Stepanov, A. & Lee, M. 1995. The Standard Template Library. http://www.cs.rpi.edu/~musser/doc.ps, Hewlett-Packard Laboratories, Nov. 11, 1996.

Stepney, S., Barden, R. & Cooper, D., Eds. 1992a. *Object Orientation in Z*. Workshops in Computing, Springer-Verlag.

Stepney, S., Barden, R. & Cooper, D. 1992b. A Survey of Object Orientation in Z. *Software Engineering Journal* **7**(2): 161-168.

Stroustrup, B. 1991. *The C++ Programming Language (second edition)*. Reading, Mass., Addison-Westley.

Sun Microsystems Inc. 1996. JavaSoft. http://java.sun.com/, September 25, 1996.

Wang, D. & Lee, J. R. 1993. Visual Reasoning: its Formal Semantics and Applications. *Journal of Visual Languages and Computing* **4**(4): 327-356.

Wegner, P. 1988. The Object-Oriented Classification Paradigm. Research Directions in Object-Oriented Programming. B. Shriver & P. Wegner. Cambridge, Massachusetts, MIT Press**:** 479-560.

West, D. B. 1996. *Introduction to Graph Theory*. Upper Saddle River, NJ, Prentice-Hall.

Wirth, N. 1988. *Programming in Modula-2*. Berlin, Springer.

Zalta, E. 1988. *Intensional Logic and the Metaphysics of Intentionality*. Cambridge, MA, MIT/Bradford.

Zalta, E. 1993. Twenty-Five Basic Theorems in Situation and World Theory. *Journal of Philosophical Logic* **22**: 385-428.