1-2007

Dynamic Networking of Exemplars: Towards a Mechanical Design Visual Programming Language

Shashidhar Putti
Clemson University, sputti@clemson.edu

Dynamic Networking of Design Exemplars: Towards a Mechanical Design Visual Programming Language

---

A Thesis
Presented to
the Graduate School
Of Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Mechanical Engineering

---

by
Shashidhar Putti
May 2007

Accepted by:
Dr. Joshua Summers, Committiee Chair
Dr. Georges Fadel
Dr. Vincent Blouin

ABSTRACT

Developing mechanical engineering design automation applications using current day textual programming languages like C, C++ requires extensive programming skills. However, learning to program in these environments can be time consuming and often frustrating to mechanical engineers due to the use of complex textual constructs and syntax to represent an algorithm in a linear fashion. Further, these programming languages offer a poor visualization of the data and the data flow due to their textual nature. The use of graphical objects for programming makes visualization easier and also eliminates the need for linear representation of algorithms. This thesis initiates the development of a visual programming language that makes use of objects representing geometric entities and relations for programming.

The design exemplar was introduced in the literature as a representation that makes use of a structured set of entities and relations for representing, querying and modifying design data. This research investigates the possibility of developing the design exemplar system into a visual programming environment for mechanical engineers. To this end, the basic components of a visual programming language are identified and compared with the design exemplar system. The aspects of a programming language that the design exemplar system currently does not support are identified which includes the basic components and programming constructs. While, a visual language compiler is identified to be essential for this new programming environment, programming constructs like looping, conditional branching are identified to be important for handling large sets of data processing operations and thus can be useful for developing design automation programs.

As a step towards the development of this visualized programming language, the design exemplar, system which inherently supports problem solving using the declarative design exemplar representation, is extended to support procedural processing of design data. This new feature in the design exemplar system extends the exemplar to support conditional branching and looping operations, which were identified as important for handling the data processing operations. The development of a compiler remains out of scope for this research and is left for the future work.

DEDICATION

To my parents, for their love and support.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Joshua D Summers, the driving force behind this work. This thesis would not have appeared in its current form without his patience, support and guidance. I would also like to express my gratitude to the members of my committee, Dr. Georges Fadel and Dr. Vincent Blouin for their continuous support and advice.

I am thankful to my friends and colleagues in CREDO lab, especially Srinivasan Anandan, Vikram Bapat, Madhusudhan Kayyar and Murthy Srirangam, etc for providing a stimulating and fun environment in which I could learn, work and grow. I would also like to thank, Jaisimha Boorugu, Chalam Tubati, Hemanth Siddulugari, Prasanth Adurthi, Arvind Kotturi, Abinand Chelikani, Kalyan Neelam and the other friends who have made my stay at Clemson, memorable.

I would specially like to thank Sowjanya Devender and Swapna Chilukuri for their continuous support.

Above all, I would like to thank my parents who made all of this possible.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES ix

CHAPTER

1. INTRODUCTION 1

   Thesis Outline 2
   Problem Statement 3
   Background 4

2. VISUAL PROGRAMMING 8

   Visual Programming Languages – An Introduction 9
   Generalized Icons (Icons) 10
   Iconic System 13
   Visual Language Compiler 14
   Important Programming Constructs: 17
   Summary 20

3. THE DESIGN EXEMPLAR 21

   Illustration 22
   Design Exemplar Algorithm 24
   Logical Connectives 27
   Exemplar Networks 28
   Summary 29

4. THE DESIGN EXEMPLAR AS A VISUAL PROGRAMMING LANGUAGE 30

   The Design Exemplar – Icons 31
   The Design Exemplar - Iconic System 35
   The Compiler 38
   Other Aspects of Programming Language 41
   Problem Identification 44

Table of Contents (Continued)

Page

5. DYNAMIC NETWORK OF EXEMPLARS 48

Dynamic Node 49
Pure Network 51
Cyclic Network 52
Dynamic Networks with Logical Connectives 57
Case Study: Gear Train Design 58
Case Study: Performance Evaluation 67

6. CONCLUSIONS 82

Contributions 84
Future Work 85

LIST OF REFERENCES 86

LIST OF TABLES

Table		Page
2.1	Constructs for Looping	19
3.1	The Design Exemplar Algorithm: Validation (Summers. 2004)	26
3.2	The Design Exemplar Algorithm: Transformation (Summers. 2004)	27
4.1	Table showing some Icons used in the Design Exemplar System and the Visual and Logical Information in them	31
4.2	Table showing a list of Object Icons in the Design Exemplar Vocabulary	33
4.3	Object Icons (representing geomertric relations) used in the Design Exemplar System	34
4.4	Iconic Operators of the Design Exemplar System	36
4.5	Physical and logical parts of the various icons in the Design Exemplar System	37
4.6	Logical information in radius relation and straight line icons	39
4.7	Table showing – Count Controlled and Condition Controlled loops	43
4.8	Algorithm for a Gear Train Design (Given, the ratio)	45
5.1	Algorithm to find the boss with the shortest radius value in Figure 5.4	53
5.2	Operations performed in each iteration of the nodes.	55
5.3	Algorithm for gear train design using exemplars	58
5.4	The Exemplars used to solve the Gear Train - Design Problem	59
5.5	Table showing the various operations performed at each node and the evolution of the gear train in Iteration 1	63
5.6	Table showing the various operations performed at each node and the evolution of the gear train in Iteration 2	65
5.7	Table showing the various operations performed at each node and the evolution of the gear train in Iteration 3	66
5.8	The line configurations considered for the Performance Evaluation Case Study	67
5.9	Results obtained with the Dynamic Networks in Case 1	73

List of Tables (Continued)

Table		Page
5.10	Results obtained with the Dynamic Networks in Case 2	74
5.11	The checks performed and the results obtained in CASE 3.	75
5.12	The pairs of line-segments identified in Case 4, Case 5, Case 6.	75
5.13	Big (O) Complexity for each algorithm	81

LIST OF FIGURES

Figure		Page
2.1	Figure showing Object Icons used to represent numeric values in LabVIEW	11
2.2	Figure showing Object Icons used to represent complex numeric data-structures in LabVIEW	12
2.3	Figure showing Process Icons used to represent numeric operators in LabVIEW	12
2.4	Visual Sentence in LabVIEW to calculate the product of two integer variables	14
2.5	A Typical Visual Language Compiler	15
2.6	Parse tree for the Visual Sentence shown in Figure 2.4	16
2.7	Conditional Statement in LabVIEW, to find the square root of a number (Jemielniak. 2004)	18
2.8	Conditional statement in C, to calculate the square root of a number	18
3.1	A model Figure showing two parallel planes P1, P2	21
3.2	Bi- partite graph representation of a pair of parallel panes	21
3.3	Sample design model to be queried	23
3.4	Exemplar to find radius of the circle -- tangent to a Line	23
3.5	Matches obtained when the two-circle-two-line model is queried using the design exemplar shown in Figure 3.4	23
3.6	Design exemplar to find a circle tangent to a line and modify its radius to 25mm in the 3.3	24
3.7	The modified design model of Figure 3.3	24
3.8	The Transformation and Validation Axes of the Design Exemplar Operation [Summers. 2004]	25
3.9	Exemplar to find a pair of lines that are either parallel or perpendicular	28
3.10	Exemplar to find a pair of lines perpendicular to each other but do not intersect.	28
3.11	Figure showing a Static Network [Summers. 2004]	29

List of Figures (Continued)

Figure		Page
4.1	Complex Icon representing a line segment	35
4.2	Visual representation of a Circle with Radius R	37
4.3	The System Diagram of the Design Exemplar Compiler extended from the SIL-ICON Compiler [Chang et. al., 1989]	40
4.4	A spring -- mass system acted upon by a force F	46
4.5	Simulink Model representing Equation (a)	46
5.1	Complete Exemplar Node for Dynamic Networking	50
5.2	Simplified Exemplar Node for Dynamic Networking	51
5.3	A Pure Network Figure showing its various levels	52
5.4	Model with Three Bosses	53
5.5	A Cyclic Network of exemplars representing the algorithm shown in Table 5.1	54
5.6	Changes the model undergoes while traversing the Dynamic Network	56
5.7	A Dynamic Network with exemplar formed using Logical Connectives	57
5.8	The Dynamic Network representing the Algorithm for the Gear Train Design using the Dividing Method:	62
5.9	Three Line Model	68
5.10	Pure Network for solving the two line problem	69
5.11	Hybrid Network to find the shortest distance between two lines	70
5.12	The shortest and longest paths followed in algorithm	78
5.13	Shortest and Longest Paths in a Hybrid Network	79
5.14	The Shortest and Longest Path in a Pure Network	80

CHAPTER 1

INTRODUCTION

Mechanical engineers often use software applications that facilitate tasks like design (SolidWorks[1]), analysis (ANSYS[2]), and manufacturing (ProCam[3]). The use of these applications is becoming more wide spread with the increasing need for faster and more accurate information in order to reduce design cycle time and to improve design quality. For example, the use of finite element analysis software like Ansys [Ansys. 2006] or Abacus [Abacus. 2006] to evaluate the stress induced in a disc brake due to thermal and structural loading is faster and often yields more accurate results when compared to using analytical methods to manually calculate them [Hu. 2005]. Specialized applications are being developed to facilitate every aspect of mechanical engineering. However, these applications are mostly developed using high level textual programming languages like C, C++. Since, in developing such applications, developers need considerable amount of software engineering expertise, which is not often found in mechanical engineers, they have to either depend on software programmers who are not mechanical engineers or mechanical engineers that have learned programming. Learning to program using textual languages usually involves learning to organize the operations to be performed on the data in a sequential fashion using syntactic textual constructs. Since, these textual constructs are usually complex and change from one programming language to the other the process of learning is often found tedious and time consuming [Shu. 1988]. In addition, the linear representation of algorithms which are not always linear makes this process even more tedious and forces the programmer to concentrate more on the program dynamics rather than the intent of the program itself [Shu. 1988] [Chang. 1990] [Ahmad. 1999].

Representing logical objects in high-level textual programming languages and visualizing the data flow requires fair amount of expertise. These factors have formed the rationale for the development of visual programming languages (VPL), where visual objects such as pictures, symbols, and graphs etc are used for programming. These programming languages offer a much better visualization of the data flow in

[1] http://www.solidworks.com
[2] http://www.ansys.com
[3] http://www.teksoft.com/

a program. Since mechanical engineers largely depend on programmers for their software needs, it is desirable to develop a visual programming language that can be used to reduce this dependency. Such a VPL should be able to make use of objects like circles, lines, cylinders and planes etc that the mechanical engineers are most familiar with. This eliminates the need to learn complex textual constructs and linear representations of algorithms otherwise required in textual programming languages, while improving the visualization of the program.

The possibility of using the design exemplar system, which has been introduced in the literature as a data structure for representing design data, for developing such a visual programming language is evaluated in this research. This research presents a comparison between the essential components of a visual programming language and the existing components of the design exemplar system and seeks to identify the components that are missing.

Thesis Outline

The information relating to the factors that led to this research, the way this research is conducted, the results, and the inferences made at the end of the research are organized into six chapters in this dissertation.

- Chapter 2: This chapter introduces the concept of visual programming, discussing various visual programming languages found in the literature. A clear distinction between textual and visual programming languages is drawn in this chapter to show the reason for considering visual programming languages for this research. The basic components of a visual programming language are identified.
- Chapter 3: This chapter presents the design exemplar and discusses the various components within it. The pattern matching, design validation, model modification, and querying capabilities of the design exemplar are illustrated. Additionally, design exemplar extensions are examined as they relate to facilitating the use of the design exemplar as a VPL. These tools include Logical Connectives [Divekar. et. al., 2004] and Static Exemplar Networks [Summers. 2004].
- Chapter 4: In this chapter, a comparison is drawn between a visual programming language and the design exemplar system to identify the components that are missing in the design exemplar system. The importance of each of these components is discussed and the issues that need to be addressed to include these components into the existing exemplar system are identified.

- Chapter 5: The concept of Dynamic Exemplar Networks as a way to achieve procedural processing of exemplars is presented in this chapter. The algorithm for this proposed new approach is presented. This provides for the significant missing components of the design exemplar system identified in Chapter 4: An illustration is provided to explain the dynamic networks approach. The validation for various hypotheses of this research is presented in this section using a case study approach.
- Chapter 6: A summary chapter concludes this thesis while outlining the contributions of this research. The issues that are identified but are out of scope for this research are presented here as the future work.

Problem Statement

The design exemplar has been introduced in the literature as a data structure for representing design data that can be used for pattern matching, querying design information, modifying and validating design. The design exemplar representation is based on geometric, topologic and parametric entities and constraints and makes use of graphical objects to represent them. This ability of the design exemplar system to make use of graphical objects for developing applications useful in mechanical design has formed the foundation for this research. This has led to the idea of developing the design exemplar system into a visual programming language. Hence, this research seeks to answer the following questions:

Q 1: Can the Design Exemplar system be developed into a visual programming language for mechanical design?

Q 1.1: What is missing from the Design Exemplar Technology to support a Mechanical Design VPL?

Q 2: Can a procedural approach be employed in a visual programming language for mechanical design?

Q 2.1: Can a procedural approach be developed to extend the exemplar to support conditional branching?

Q 2.2: Can a procedural approach be developed to extend the exemplar to support looping?

Q 2.3: How does the new approach affect the complexity of solving?

Q 2.4: How does the new approach affect the number of solutions obtained?

Background

The advent of CAD systems has facilitated representation and visualization of engineering design [Groover. et. al., 2003]. However, generating the product models within these CAD systems has largely been a manual process, which depended on factors such as the knowledge or experience of the designer. This knowledge was typically not explicitly passed from designer to designer [Agarwal. et. al., 2000]. While this manual design process can be useful for developing intuitive and novel designs that are not possible to produce using the set of pre-defined rules, it is slow because of human information processing, exchange, and reasoning [Agarwal. et. al., 2000] [Starling. 2004].

Automating the various aspects of the design cycle has been identified as a way to reduce this delay and hence reduce the design cycle time for design problems that are routine in nature [Gero. et. al., 1998]. Rule based design has been used to automate routine design in mechanical engineering. Two different approaches to rule based design include expert systems (rules are fired as their condition become active) [Ullman et. al., 1989] and production systems (rule sequencing is predefined) [Engelke. 1987]. Production systems [Post. 1943], largely used in chip design and architecture [Stiny. et. al., 1978] [Koning. 1981], have been used also for the development of mechanical designs like robots [Lipson. et. al., 2000]. Production systems are domain specific, where the knowledge base required for solving a problem within the domain is represented in the form of rules. These rules denote constraints that enable procedures to seek new assertions or to validate a hypothesis. These rules represent constraints that enable to validate a condition or to seek new assertions [Winston. 1993]. These rules are fired in a procedural fashion to create a design that satisfies the given set of requirements. Production systems traditionally are used for manipulating symbols and abstract entities representing logical data using the rules defined within the knowledge base of the system. Representing geometric entities and relations would typically require a huge rule base, which is typically considered beyond practical, hence most productions systems used for mechanical design applications represent the design data in the form of abstract entities and relations. A popular use of production systems for mechanical design applications is in the geometric manipulation of product form [Agarwal. et, al., 2000] [Stahovich. 2001] [Forbus. et. al., 1991].

Agarwal [Agarwal. et al, 2000] has used shape grammars for the generative design, while overcoming the shortcomings pertaining to the level of geometry and rule base size seen in the production system approach mentioned above. Formally, shape grammar is defined as a 4-tuple of G = (V, X, R, S) [Starling. 2004], where the V is finite set of shapes; X a finite set of symbols; R is a finite set of rules of the

form a → b, where a and b are defined to be valid for the set of (V, X); and S a labeled shape in (V, X) called the initial shape [Agarwal. et. al., 200]. Shape grammars employ a generative design methodology of match and transform operations based on the defined set of shapes and shape transformation rules [Gips. et. al., 1980]. The use of shape grammars in solving various engineering design problems like dome design, design of roof trusses, form generation of motorcycles and coffee makers have been presented in [Shea. et. al., 1997] [Shea. et. al., 1999] [Pugleise. et. al., 2002] and [Agarwal. et. al., 1999] respectively. However, shape grammars are developed domain specific and are developed to solve specific problem [Deak. et. al., 2006]. In addition, the lack of a generalized grammar to represent the problems of a domain makes it difficult for others to understand, not associated with the grammar formulation. The sequencing of rules is not explicitly predetermined, posing the problem of not guaranteeing that a valid design can be found. For example, the rules to design a gear train, when fired in a wrong order may lead to incomplete or designs that do not server the purpose. To avoid such a situation, the use of shape grammars for design synthesis where the sequence of rule application is explicitly defined is presented in [Liew. et. al., 2000].

Design automation tools that rely on rule based approaches have been developed for a broad range of applications from ball joints, actuators [Lipson. et al., 2000], design of truss structures [Reddy. 1995], [Shea. 1997]. These tools have been developed by programmers and engineers working together in a high level language. The need for a better programming environment for mechanical engineering applications apart from the various issues pertaining to the representation, usability, or logical generation of design have formed the reasons behind the development of programming languages specific for geometry and structures. The development of these programming languages has widely been researched.[Laakko. et. al., 1993] [Milicchio. 2005] [Banyasad. et. al., 2001] [Heisserman. 1993] [Nackman. 1986]. However, most of these programming languages have been developed as an extension to an existing programming language and, hence, the representation and the logic of programming is inherited from the original programming language. This section reports the various programming languages found in the literature.

Sketchpad [Sketchpad. 1963] can be considered as a first step towards the development of constraint programming languages. Though it was developed as a general-purpose system for mechanical, electrical, scientific drawings, it allows engineers to interactively build geometric objects from language primitives such as lines or arcs. Sketchpad stores the explicit information pertaining to the topology of the geometry and automatically shows the variation in the entire geometry, when a parameter is changed. Sketchpad also provides the user with the ability to develop new constraints and geometric primitives that

can be used for programming. Subsequently, languages for constraint programming like THINGLAB [Borning. 1982], JUNO [Nelson. 1985], CONSTRAINTS [Steele. et. al., 1980] have been developed primarily for constraint solving, while using textual or graphical primitives.

Language for structure design (LSD), a high level visual logic programming language for design of structured objects, was developed based on a PLaSM kernel, which in itself is a functional programming language and is used for maintaining a low level description of the solids and operations. Developed by Banyasad and Cox [Banyasad. et. al., 1997], this language was built by extending the grammar supported by LOGRAPH, a general-purpose visual logic programming language. While the main intent for developing it was to support designing parameterized components of complex structures in a homogeneous environment, it also provides the capabilities for programming [Banyasad. et. al., 2001]. Since LSD inherits logic-programming problem-solving abilities from LOGRAPH, it also may be used to address design specification issues such as searching. However, the representations used for programming or developing design specifications in this environment are inherited form LOGRAPH and do not provide an intuitive representation of the geometries or solid objects involved. LSD supports four basic operations: replacement, deletion, merging, and bonding that can be preformed on e-components (explicit components) and i-components (implicit components) to build an assembly according to the designs in a program [Banyasad. et. al., 2003]. Again, it should be noted that these operations are not directly performed on solid objects, instead, are performed on the cells and cases, which provide an abstract representation of the low-level solid objects and operations in PLaSM. An assembled design specification is an anchor-knot network that is then translated into a PLaSM program using a translator. This anchor-knot network is represented as a list of function definitions providing a description of the solids and operations. From the representation of objects and logic in this language, it is evident that the user is expected to be familiar with LOGRAPH, a visual logic programming language. Since, such a programming language will largely be used by mechanical engineers; this is an entirely new domain and poses new challenges.

Shu [Shu. 1988] classified visual languages, based on the objectives of these languages, into those for (a) handling visual information, (b) supporting visual interaction, and (c) programming with visual expressions. It is the third kind of language with which this research deals. The symbols used for visual programming are called graphical objects, each of which is characterized by a set of graphic, syntactic, and semantic attributes [Costagliola. et. al., 2002]. These attributes hold the information relating to the appearance, position, and the way that graphic object is related to the other graphic objects [Costagliola. et.

al. 2002]. These graphical objects form the primary building blocks for visual programming, meaning that they are combined to form meaningful visual sentences to serve a specific programming task. The graphical attributes of the objects describe the color, shape, size, location, name, while each object has a predefined set of attaching points as syntactic attributes. These points are in turn connected through poly-lines that visually depict the control flow relation among objects. The semantics associated to the attaching points of these graphical objects defines the direction of the connections. The type of syntactic attributes and the types of feasible relations that can be applied to them to compose visual sentences are strongly related and characterize a visual language class [Costagliola. et. al., 2002].

CHAPTER 2

VISUAL PROGRAMMING

In the past forty years, programming languages have developed from low level languages like the machine and assembly languages to high-level languages like C, C++, or FORTRAN [Chang et. al., 1989] [Shu. 1988]. While the motivation behind this development has been to ease the task of programming, the tradition of linear representations, where the input instructions are given in a statement-by-statement fashion, has persisted. However, the input details required from the user have reduced from generation to generation [Chang. 1990]. For example, programming in a high-level language like C or C++ deals with complex arithmetic and Boolean operations on data variables, while programming with low-level language not only deals with manipulation of data but also with memory addresses and registers.

Once the requirements of a program are known, programming typically involves problem analysis, charting, coding, and testing [Chang. 1990]. The requirements of the program are first analyzed for scope or feasibility and by listing all possible questions pertaining to the scope, feasibility allocation of resources, or time. Once, these questions are answered, the process of planning the project execution starts. This typically involves designing the program structure and allocating time and resources. Tools like, flow charts are used to represent these structures. The algorithm is then implemented and tested against some test cases to evaluate the program functionality, in the coding and testing phases. While each of these phases is believed to be essential for developing a comprehensive code, the two phases of charting and coding seem to consume the largest amount of time. This is due mostly to the fact that the end users are not usually programmers themselves and need to learn programming before using it [Shu. 1988]. For example, the development of a software application to automate a manufacturing or design process could require a mechanical engineer to learn programming in a text based high level language environment or to spend significant time explaining and detailing the principles involved with the specific application to a software developer whose primary concern is software code writing and optimization. Textual programming languages typically require using textual constructs, like the keywords for declaring variables, or performing checks based on a predefined syntax for performing data manipulation operations.

The data in such programming languages is represented in a textual format making it difficult to visualize the data flow. Irrespective of the structure of an algorithm, the textual nature of the representation in these programming languages limits them to be represented in a linear fashion [Chang. 1990]. These factors often make the process of learning to program time consuming and tedious. It may be noted that while the main aim of programming is to logically manipulate data, most of the time spent while programming using a textual programming languages is spent on implementing the logic of programming [Shu. 1988].

Visual Programming Languages – An Introduction

The fact that people think and relate to the world in terms of graphic imagery has laid the foundation for the thought that using meaningful graphical objects to construct programs would ease the task of programming [Boshernitsan. et. al., 2004] [Chang. 1990]. This in turn has paved the way for the development of visual programming languages (VPL), the underlying motive of which is to make programs and programming more readable, understandable, and less error prone . Though this is difficult to prove with comprehensive empirical evidence, it is often seen and understood that the use of graphic symbols instead of textual constructs makes the process of visualization easier [Shu. 1988]. The motivation behind the visual programming language is to develop a goal oriented approach rather than concentrating more on developing the code required to implement the goal [Frasson. et. al.,]. This type of programming language is classified as a fifth generation language. The following are a few definitions of a visual programming language found in the literature:

Shu [Shu. 1988] informally defines visual programming language as "*A language which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language*". Chang [Chang. 1990] defines visual programming language as *"a set of iconic sentences constructed with given syntax and semantics"*. Lu [Lu. et. al, 1988] formally define visual language as a graph representation which can be generated by a visual grammar G_v, where G_v is a four-tuple of sets of complex visual objects ($\boldsymbol{V_N}$), primitive visual objects ($\boldsymbol{V_T}$), production rules (***P***), and starting symbols (***S***).

L_v = {x|x is a graph representation which can be generated by a visual grammar G_v}

$$\boldsymbol{G_v=(V_N, V_T, P, S)}$$

Shu's [Shu. 1988] classification of visual languages based on their objective use was presented in Chapter 1. Since, the motivation for this research is to reduce or eliminate the need for textual constructs and to use objects that can make programming easier for mechanical engineers, while providing visual interaction, this research deals with the third kind, namely "programming with visual expressions", presented in this classification [Shu. 1988]. Certain aspects like the vocabulary, grammar and the type of compilers used determine the level of input detail and the expertise required for programming in a visual language environment. These aspects are presented in the following sections using, "G", a visual programming language developed by National Instruments for data acquisition, instrumental control and industrial automation as an example [LabVIEW. 2006].

Vocabulary and Grammar

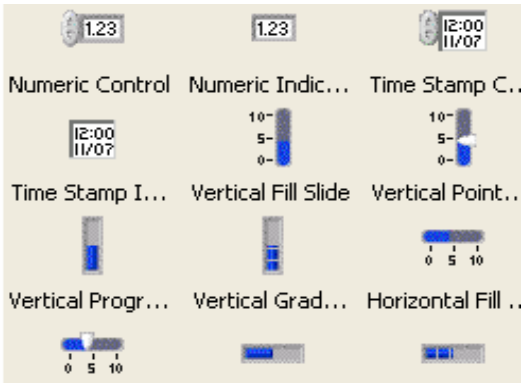
The symbols used for visual programming, graphical objects, are characterized by a set of graphic, syntactic, and semantic attributes [Costagliola. et. al., 2002]. These attributes hold the information relating to the appearance, position, and the way that graphic object is related to the other graphic objects [Costagliola. et. al., 2002]. These graphical objects form the primary building blocks for visual programming as they are combined to form meaningful visual sentences to serve a specific (programming) task. The graphical attributes of the objects describe the color, shape, size, location, name, while each object has a predefined set of attaching points as syntactic attributes. These points are in turn connected through poly-lines that visually depict the relation among objects in a declarative programming language, while they represent the direction of control flow in a procedural programming language. The semantics of an object like the type of data, associated with the icon, defines the direction of the connections.

Generalized icons, an iconic system, and a visual language compiler are identified as the basic components of an iconic visual programming language [Chang. 1990]. The vocabulary and grammar of such an iconic visual programming language is characterized by its icons and iconic operators [Chang. 1990]. A syntactic combination of the icons using the iconic operators is governed by a pre-defined set of rules defined in the iconic grammar. These basic components of a visual programming language are discussed in detail, in the following sections.

Generalized Icons (Icons)

Visual languages are developed based on the concept of generalized icons, sets of icons representing the data/objects useful for programming. However, the ease of creating icons to meaningfully represent

objects of physical importance and the size of the icon set that is required to generally represent all the possible objects determines the usability of this concept in the development of a specific visual programming language. Consider for example the domain of gear design where there is a finite vocabulary (types of gears, relationships between gears) and the domain of function structure modeling where there is not a commonly recognized finite vocabulary to describe mechanical functionality. In the first example, a finite vocabulary for gears can meaningfully represent comprehensively this design domain, but in the second, larger design domain, the difficulty of representing all the possible functions using icons makes the development and usability of such a VPL difficult. The generalized icons are further divided into object icons and process icons / operator icons based on the functionality [Chang. 1990]. While object icons are used to represent logical data like numerical values in a program, the process icons or the operator icons are used to represent constructs that can be used to manipulate the data in the object icons. These object icons and operator icons are combined appropriately to express the algorithms. For example, a few object and process icons for arithmetic variables and operations in LabVIEW [LabVIEW. 2006], a platform for visual programming, generally used for data acquisition, industrial automation and instrument control, are shown in Figure 2.1, Figure 2.2 and Figure 2.3. These examples are explained below.



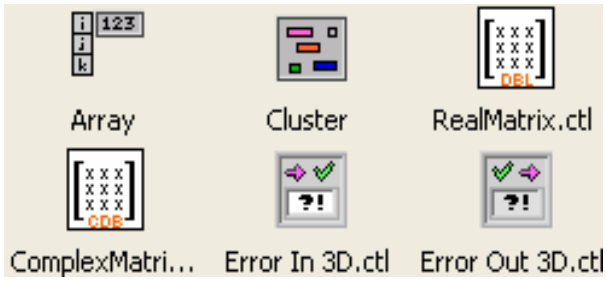
(a) Tool bar with icons to represent numeric values in LabVIEW

(b)

(c)

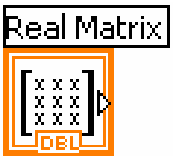
(d)

Figure 2.1: Figure showing Object Icons used to represent numeric values in LabVIEW



(a) Tool bar with icons to represent complex numeric values in LabVIEW

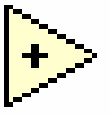
(b)



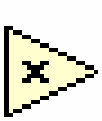
(c)

Figure 2.2: Figure Figure showing Object Icons used to represent complex numeric data-structures in LabVIEW

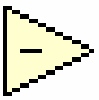
(a)



(b)



(c)



(d)

Figure 2.3: Figure Figure showing Process Icons used to represent numeric operators in LabVIEW

Figure 2.1(a) shows the tool bar holding the icons for numerical objects and Figure 2.1 (b-d) shows the various ways of representing a numerical value in LaBVIEW, Figure 2.2 (a) shows the tool bar for object icons that can be used for representing a data structures holding numerical data. For example, an a matrix of real numbers, which are formed by arranging a set of real numerical values in rows and columns may be represented using the icons shown in Figure 2.2 (c). A generalized icon shortly called as an icon, besides providing a graphical representation of an object, also holds the information or the data associated with that object. The icons shown in Figure 2.1 (b-d), Figure 2.2 (b, c), apart from serving the purpose of visually representing the numeric variables, also hold the numerical values associated with them. These values are required for the arithmetic processing in the programs in which they are used. Visual programming languages allow the manipulation of the information within these object icons by operating on them using the process icons defined within the visual language. A few process icons used to represent arithmetic operations in a LabVIEW program are shown in Figure 2.3 (a-d). Figure 2.3 (a) shows the tool bar for process icons used to represent arithmetic operations. For example, while Figure 2.3 (a) shows a process icon used for representing an arithmetic addition operation, the process icons shown in Figure 2.3 (b) and Figure 2.3 (c) represent arithmetic multiplication and subtraction operations

Depending on the level of abstraction offered by an icon, they may be classified into either elementary or complex icons [Chang. et. al., 1990]. Icons that represent the basic entities or the primitive objects used for programming are called primitive icons, while icons that are formed by syntactically combining several primitive icons of an iconic system to represent a complex object of programming are complex icons. The elementary icons are analogous to primitive data types like integers, floats, or characters in a textual programming language. These icons cannot be further decomposed or expressed as a combination of other icons. On the other hand, complex icons are formed by combining the icons of the iconic system using iconic operators and are analogous to data-structures or classes in a textual programming language like C++.

Iconic System

A set of generalized icons (Object icons and Process icons) combined with the necessary iconic grammar form an iconic system [Chang. et. al., 1990]. The icons in an iconic system may be appropriately arranged to form a visual statement representing the desired programming task. Such a spatial arrangement of icons to depict a programming task or algorithm is called a V*isual Sentence* [Lakin. et. al., 1987]. In

other words, a *Visual Sentence* [Lakin. et. al., 1987], also called as *Iconic Sentence* [Chang. et. al., 1990], *Iconic sentence* [Tanimoto. et. al., 1986], or I*conic statement* [Korfhage. et. al., 1986], is constructed bya a spatial arrangement of clearly defined object and process icons defined in the iconic system. The complex icons discussed earlier may be seen as a special case of the visual sentences, where the iconic sentence as a whole is used as an icon. Again, the complex icons may be further connected syntactically to form meaningful visual sentences that represent the desired algorithm. Figure 2.4 shows a visual sentence to calculate the product of two integer variables A, B and assign the value to a variable C, composed in LabVIEW [LabVIEW. 2006]. Two object icons representing the integer variables A, B are connected to the input of the process icon for multiplication. The output of this icon is in-turn connected to an object icon representing integer C. The directions of the arrows on these icons represent the direction of control flow. Hence, the numerical value assigned in the icons representing variables A, B are multiplied and the value is stored as logical information in the icon representing variable C.

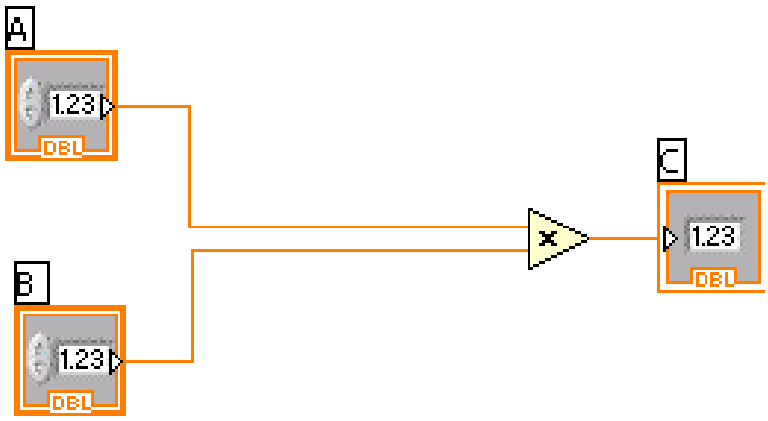


Figure 2.4: Visual Sentence in LabVIEW to calculate the product of two integer variables

The authoring of visual sentences by combining object and process icons syntactically is governed by rules defined with-in the iconic grammar.

Visual Language Compiler

The set of generalized icons (Object icons and Process icons) combined with the necessary iconic grammar form an iconic system [Chang. et. al., 1990]. Visual programming languages, categorized into the fifth generation programming languages, have evolved to facilitate the process of programming, by providing a better visualization of the data and data flow [Shu. 1988]. Such languages allow the user to

operate at a level of abstraction that does not require handling with registers and memory addresses [Kay. 1984]. Hence, these programming languages provide for a better understanding of the program making it easier to use for the programmer. However, the programs composed or written in such languages are translated into source code in a target language, usually a lower level language such as assembly language that can be interpreted by the computer. This conversion of the source code into an execucode or the object code is performed by the compiler of the visual programming language. Hence, the compiler is regarded as an important component of a programming language. Figure 2.5 shows the structure of a typical visual language complier.

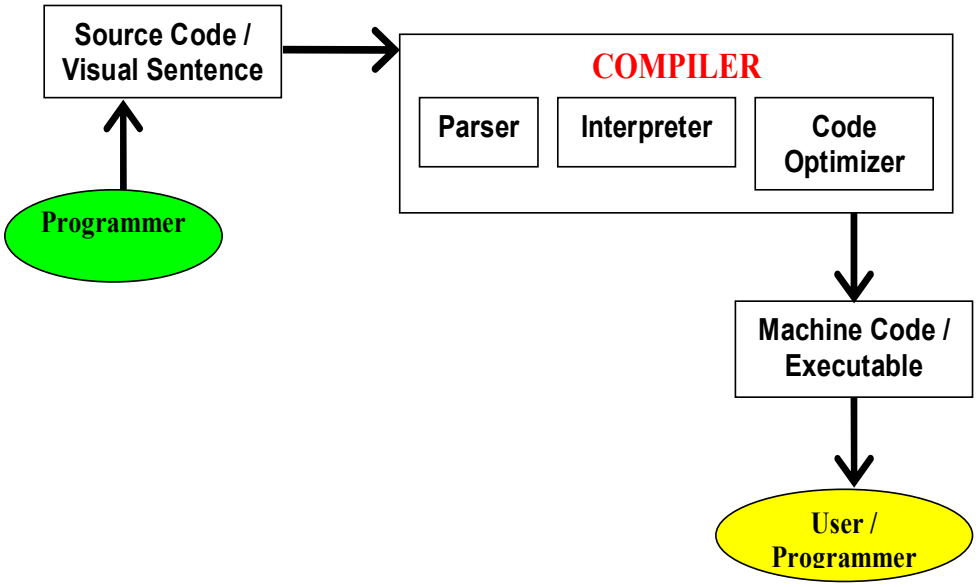


Figure 2.5: A Typical Visual Language Compiler

Compilers first parse the visual sentences and then perform a semantic analysis of the parsed representation followed by code optimization and then source code generation. Parsing typically involves the generation of a data-structure, called the parsing tree, that describes the hierarchy of the input data and which can be used for further processing. The grammatical structure of the input sequence is checked for correctness and validity with respect to a formal language grammar. In a compiler for a visual programming language, this analysis is performed based on a formal graph grammar or iconic grammar. As an example, the syntactic analysis in a SIL-ICON compiler [Chang. 1990] is performed based on an extended task action grammar (ETAG) [Chang. et. al., 1989]. Semantic analysis can be supported by different types of grammars, such as picture grammars [Chang. et. al., 1971], precedence grammars [Chang

et. al., 1970], and graph grammars [Fu. 1974] depending on the nature and domain of the visual programming language.

Secondly, semantic analysis involves adding semantic information to the data-structure, parsing tree, obtained after the parsing phase is completed. This includes operations like type checking, variable binding, or object binding, where the use of the various objects and variables is checked to see their consistence with the definition. Semantic analysis generates an intermediate code with semantic information, which is used as the input code for the code generation phase [Ullman. et. al., 1976] [Sethi. et. al., 1986].

The compiler to a visual programming language is expected to perform all the operations mentioned above. However, since the input is in the form of visual objects, the syntactic and semantic analysis performed in this case is expected to be based on a formal visual language grammar related to its domain. These various operations performed in a visual language compiler are illustrated here with an example. A visual sentence composed in LabVIEW to calculate the product of two integers as shown in Figure 2.4 is considered for this example [LabVIEW. 2006]. When this program is compiled, the parser separates the variables (A, B, C) from the reserved characters like the symbols for multiplication here. The second stage involves the identification of variables and constants in the program and designating them as tokens. Here, the variables A, B, C are designated as variable tokens and then arranged in the parsing tree to preserve their relationship to each other. At the end of these parsing operations a data-structure called the parse-tree is obtained as show in Figure 2.6. The compiler uses on of pre-order, post-order and in-line algorithms to traverse and evaluate expression in this parse tree.

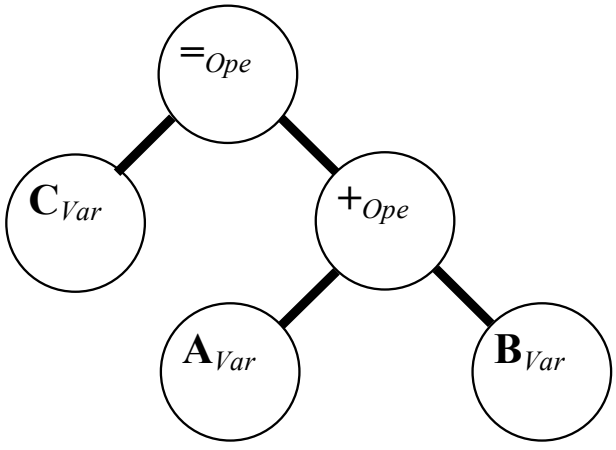


Figure 2.6: Parse tree for the Visual Sentence shown in Figure 2.4

For example, when the algorithm for in-line traversal is used to evaluate the expression in this tree, where the order of traversal is *left child – root node – right child*, the evaluation of the expression starts at the node with variable C, propagating into the right side of the tree through the *"="*node. Here, since the + node is at the same precedence level as the = node, the + node is considered the root node and the nodes *A,* + and *B* are evaluated in that order. Hence, the expression in this tree is evaluated by traversing the nodes with *c, =, a, +, b,* in that order. Finally, the compiler runs through this parse tree generating the native instructions that the CPU can directly execute.

Important Programming Constructs:

The basic components of a visual programming language, presented above, are identified to be, if not necessary, important for developing applications using a programming language. However, since present day programming often involves logical handling of large data, it is important that the programming languages support means of easily organizing and handling it. In order to facilitate this, most current high-level programming languages support methods and constructs for grouping or handling data easily. For example, data-structures and arrays found in high level programming languages like C, C++ facilitate handling of data by grouping related data. On the other hand, while constructs for iterative processing of data like *FOR*, *Do-While* statements found in these high level programming languages(C, C++) ease the task of programming by providing a way to execute a set of statements automatically until a condition is met, constructs like *if- then- else* provide a way for dynamically selecting a set of operations to be performed The ability of a programming language to support such constructs is sought to be important and is presented below in detail.

Data Grouping

The grouping of related data makes accessing and hence manipulation of said data easy. For example, the information of an employee like the name, address, or employee id may be grouped or bound together to make accessing it easier. This is an example data binding of object oriented programming thus forming the background for the use of classes in high-level textual programming languages. Furthermore, visual programming languages provide forming generalized icons similarly to data structures analogous to lists, stacks, queues, trees, or graphs. Both, primary and complex icons, as proposed in [Chang. et. al., 1989], can be used for this purpose. A group of logical data can be represented in the form of an icon, where the information required for logical processing is held in its logical part and the accommodates the

information related to the logical processing and the physical part of the icon can be suitably composed to represent this data. Similarly, complex icons can also be used to group logical information represented in the form of primary icons. However, since complex icons can be formed only by appropriately combining icons using process icons, only the data that can be logically combined can be represented in this format. For example, a complex icon to represent a rectangle can be formed by logically combining four line segment icons with angle icon operators. On the other hand, an employee class in C++, holding the name, employee id and address of an employee, where these data elements (name, employee id and address) cannot be logically related using process icons, cannot be represented using a complex icon.

Conditional Statements and Conditional Branching:

The use of conditional statements is useful when the choice of executing alternate sets of statements based on the results of a check is to be made. The statement that performs the check is called a conditional statement, while the process of choosing a set of statements based on the results of this check and executing them is called conditional branching. For example, Figure 2.7 and Figure 2.8, show a visual sentence and textual representation of a conditional branching operation.

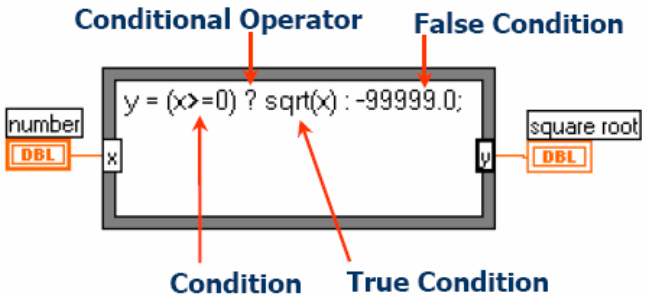


Figure 2.7: Conditional Statement in LabVIEW, to find the square root of a number (Jemielniak. 2004)

```
if (x >= 0)
        y= SQRT (x);
else
        y= -9999;
```

Figure 2.8: Conditional statement in C, to calculate the square root of a number

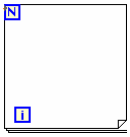
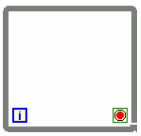
Figure 2.7 shows a visual sentence composed in LabVIEW to calculate the square root of a number. In this example, the numerical value in the variable x, the square root of which needs to be calculated, is represented by a numerical indicator *number* and the value of its square root, stored in the variable y, is indicated by the numerical indicator *square root*, while the condition statement is embedded into a *formula node* . Inside the *formula node*, the variable x is first checked to be greater than or equal to zero. If this condition is true, then the square root is evaluated and assigned to y. Otherwise, the value of the variable y is assigned -9999 [Jemielniak. 2004]. Figure 2.8 shows the textual representation of the

same conditional branching operation in a textual programming language like C. Here, the variable x is first checked in the *'if'* part of the *if-else* construct to see if its value is greater than or equal to zero. If the value of x is positive or zero then this check returns true and the value of its square root is evaluated using the *SQRT()* function (defined within the *math.h* library) and assigned to the variable *y,* in the next line of the code. If the result of the check performed in the '*if*' part returns false, the variable y is assigned a value of -9999 in the *'else'* part of the *if-else* construct used here for conditional processing. The use of a the *SQRT()* function without checking for the sign of the variable *x* can yield undesirable or invalid results (when x is negative). Hence, the ability of a programming language to support conditional statements is found useful to eliminate / reduce undesired outputs.

Looping

When a set of statements are executed iteratively until a condition holds true, the process is called looping. Looping can be of two types: pre-conditional and post-conditional. The iterative execution of a set of statements after evaluating a conditional statement is pre-conditional looping where there is a limit on the number of iterations, while the execution of the conditional statement after executing the statements to determine if the loop needs to be executed again is post-conditional looping. The left column of Table 2.1 shows the constructs used for looping through a set of operations in a textual programming language, while the right column of the shows the same in LabVIEW [LabVIEW. 2006], a VPL.

Table 2.1: Constructs for Looping in C++ and LabView

C++	LabVIEW
For (initial value; condition; increment) { CODE }	
While(condition) { CODE **}**	

A closer observation at the working of these programming constructs suggest that the execution of statements in both, conditional branching and looping is done in a sequential fashion. This suggests that the programming languages that support these constructs should support procedural execution of

statements. This can be further extended to visual programming languages. The ability of a programming language to support looping eliminates the need to repetitively specify a set of statements, thus reducing the length of the program, which in turn makes it more readable and easy to understand.

Summary

This chapter has identified generalized icons, iconic system and visual language compiler as the three most important components of a visual programming language based on [Chang. 1990]. Each of these aspects is discussed in detail highlighting the important characteristics of each. Programming constructs for data grouping, conditional processing and looping, found in most high level programming languages are identified as important to ease the task of programming. These constructs, in context to visual programming languages is discussed in detail with appropriate examples.

CHAPTER 3

THE DESIGN EXEMPLAR

The design exemplar is a data structure used for representing design data that is coupled with a generic constraint solving algorithm which facilitates querying, validation, and model modification [Bettig. et. al., 1999]. Geometric, topologic, and parametric entities and relations that are explicitly found in design models or may be implicitly inferred are represented as patterns or bi-partite sub-graphs. These patterns, or design exemplars, are used for identifying, modifying, and validating the design features of a model and reasoning information about them. In the bi-partite graph representation of model used as design exemplar, the entities and relations, vocabulary used to describe a design model, form nodes of two different groups [Summers. et. al., 2004]. A node from one group can only be connected to a node of the other using connections called edges, meaning that two or more entities can only be connected through a relation node. The two edges and the relation node joining the two entities represent the relationship between these entities. For example, a model with a pair of parallel planes, as shown in Figure 3.1 can be queried using a bi-partite graph, shown in Figure 3.2. In this bi-partite graph, the nodes representing the two entities (Plane (P1) and Plane (P2)) of the model are grouped together (shown to the left in Figure 3.2) and the node representing the parallel relation forms another group (shown to the right Figure 3.2). The two nodes representing the two entities are connected through a parallel relation using two edges, meaning that the relation between the two planes P1 and P2 is parallel.

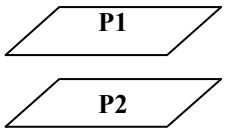


Figure 3.1: A model Figure showing two parallel planes P1, P2

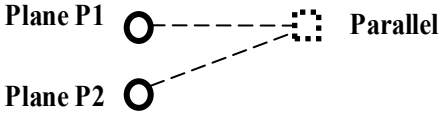


Figure 3.2: Bi- partite graph representation of a pair of parallel panes

The implicit and explicit information in a design model can be represented using the match and extract patterns supported by the design exemplar representation [Bettig. et. al., 1999]. This aspect of the design exemplar is useful for finding a feature or querying information pertaining to a feature in a design

model. The match portion corresponds to the features of the design that are of interest that explicitly exist in the design model being interrogated. The extract portion of the design exemplar is used to find or evaluate the information related to a feature that may be implicitly inferred from the matched sub-graphs. The ability of the design exemplar to modify a design comes from its alpha/beta representation as described in [Summers. 2004]. In this representation the entities and relations in the initial and final states of the model are captured as alpha and beta sub-graphs of the design exemplar. The alpha part of the design exemplar represents the valid entities and constraints of the model before a modification is performed while the beta part represents the valid entities and constraints after the modification. The modification of a model implies the conversion of the entities and relations of the model from a alpha state to the beta state or vice-versa. This transformation may include adding information, such as entities or constraints, to a model, deleting information, or changing existing information, such as changing the location or values of entity variables. This transformation aspect of the design exemplar distinguishes it from other traditional graph based feature representations.

Illustration

The following illustration in addition to serving as an example for the design exemplar representation also demonstrates the match/extract feature of the design exemplar. Figure 3.3 shows a model with two lines and two circles. A design exemplar, as shown in Figure 3.4, can be authored to identify a circle such that it is tangential to a line and then, find its radius. When the two-line-two-circle model shown in Figure 3.3 is queried using this design exemplar, the model is first checked for the match part of the exemplar, here, the line and circle, represented using thick black lines in Figure 3.4. Since, this exemplar has two match- components, a circle and a line, this check returns four possible matches, shown using thick black lines in Figure 3.5 (i - iv). These matches are then checked for the validity of the tangency constraint between the identified circle and the line. Since, the tangency constraint is valid only in the configuration shown in Figure 3.5 (iv), the circle and the line are identified identified, highlighted and the radius of the highlighted circle (shown in thick black line) is obtained.

In this exemplar, the explicit data of the desired circle-line configuration, that is, the circle and the line is represented in solid lines, whereas the implicit or the extract part of the model like the tangent constraint, the radius of the circle and the id tags are represented in dashed lines. The id tags (*id "circle"*

and *id "line"*) are useful in highlighting the identified line-circle model while the *id "radius"* tag displays the value of the circle radius.

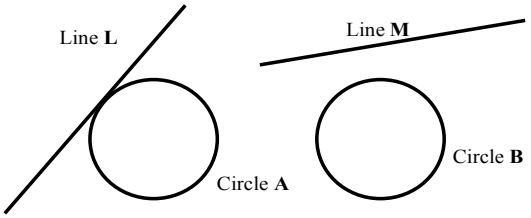


Figure 3.3: Sample design model to be queried

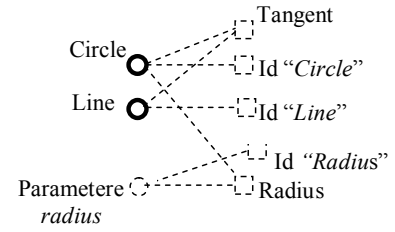
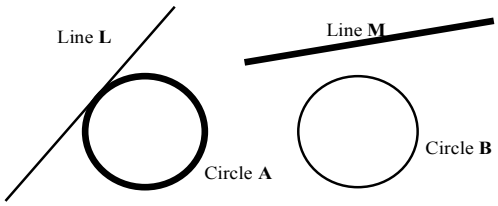
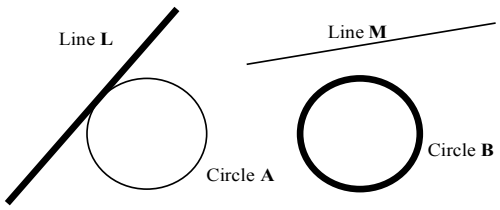


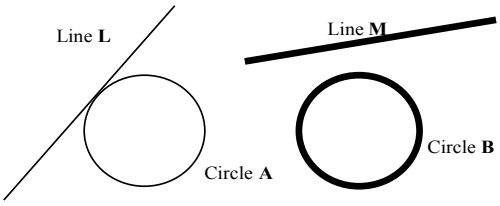
Figure 3.4: Exemplar to find radius of the circle -- tangent to a Line



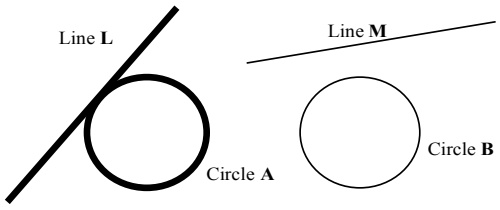
(i)



(ii)



(iii)



(iv)

Figure 3.5: Matches obtained when the two-circle-two-line model is queried using the design exemplar shown in Figure 3.4

This example is extended further to demonstrate the ability of the exemplars to modify a design. This capability of the design exemplar comes to it by the virtue of its ability to represent the design data in alpha/beta states as discussed in [Bettig et. al., 2000]. In the model shown in Figure 3.3 it is required to find the circle that is tangent to a line and modify its radius to 25 mm. In order to accomplish this task, it is required to identify the entities and relations that exist before and after the modification. The entities, circle A, circle B, line L and line M, and the implicit tangency constraint between the line L and circle A, remain before and after the modification is done and hence they exist in both alpha and beta states in the exemplar representation. Modification involves introducing a new condition on the radius parameter (to change its

value to 25 mm). Since, this condition exists only in the final state of the model; it is present only in the beta state of the exemplar representation. The design exemplar that may be used for this purpose is shown in,Figure 3.6 while Figure 3.7 shows the two-line-two-circle model after the modification.

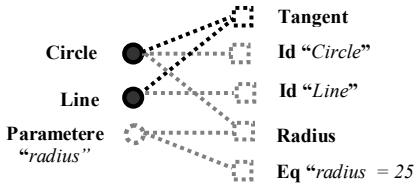


Figure 3.6: Design exemplar to find a circle tangent to a line and modify its radius to 25mm in the Figure 3.3

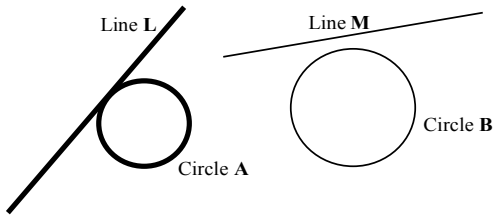


Figure 3.7: The modified design model of Figure 3.3

For clarity, the entities and relations that exist in both alpha and beta states are shown in black in the Figure 3.6 and the features that exist only in the beta state are shown in grey. Since the circle and line entities are tagged in the exemplar, these entities are highlighted in the model (shown in thick black lines), as shown in Figure 3.7. The capability of the design exemplar to modify a design by representing the alpha and beta states of a model can be used to add or delete features to a design. The declarative nature of the design exemplar representation allows the user to represent entities and relations associated with a problem without imposing the sequence of solving *a priori*. However, it should be noted that the design exemplar cannot add and modify a feature of a design at the same time. These two operations should be carried out separately in two steps using two exemplars, one to add a feature and the other to modify this new feature.. Since the design exemplar system currently does not support a way to automatically perform a sequence of operations, these two operations should be done manually.

Design Exemplar Algorithm

The design exemplar as proposed in [Summers. 2004] operates along two axes: (1) the validation axis, where a characteristic of the design model is validated to be true either explicitly or implicitly and (2) the transformation axis, where the design model is transformed from one form to the other. The match and extract parts of the design exemplar lie on the validation axis while the alpha/beta parts lie on the transformation axis. Figure 3.8 shows the graphical representation of these axes as represented by summers [Summers. 2004].

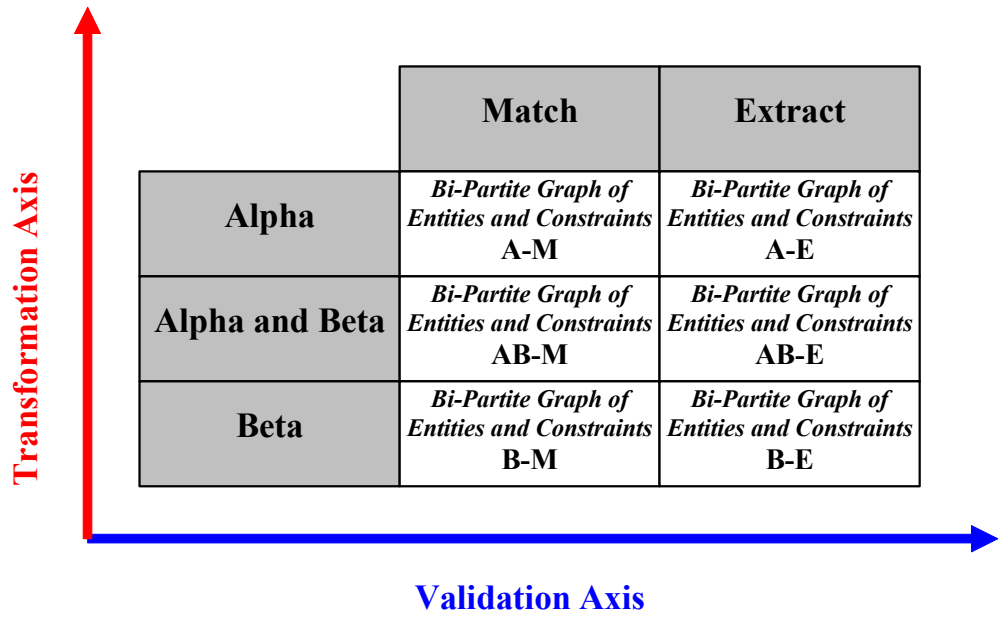


Figure 3.8: The Transformation and Validation Axes of the Design Exemplar Operation [Summers. 2004]

The design exemplar, when combined with a generic entity-relation problem solving algorithm, can be used for representing geometric and parametric design problems. The validation and transformation algorithms for the design exemplar as explained by Summers [Summers. 2004] to support the changes made with respect to transformation axis of the design exemplar (alpha, beta, alpha_beta) are shown in Table 3.1 and Table 3.2. The design exemplar algorithm creates a series of ER (Entity-Relation) problems and submits them to a set of ER solvers. The validation axis is first evaluated according to the algorithm shown in Table 3.1 and when a match is found, it is displayed. The validation algorithm is evaluated cyclically to identify all the possible matches, hence enabling the user to select the match on which the transformation needs to be applied. These algorithms are generic and hence can be applied to any domain.

Table 3.1: The Design Exemplar Algorithm: Validation (Summers. 2004)

I. Create a *Pattern Match* Entity Relation Problem
   a. Select the A-M and AB-M sub-graphs as the desired "pattern"
   b. Select the design model as the "working model" (fixing all variables so that they may not be modified in this entity-relation problem)
   c. Submit the "pattern" and the "working model" to the generic entity-relation problem solving system as a *Pattern Match* problem type
   d. If at least one "match" is found, then continue, otherwise exit as failed

II. Create a *Validation* Entity-Relation Problem
   a. For each "match" in the "working model" fix all the variables (entities) so that they may not be modified in this entity-relation problem.
   b. For each "match" in the "working model" apply the associated relationships of the A-E and AB-E sub-graphs.
   c. For each "match" in the "working model" create any entities found in the A-E and AB-E sub-graphs. Also, create remaining relationships between newly created entities and other entities.
   d. Submit the augmented "match" to the generic entity-relation problem solving system as a *Validation* problem type. This will check to ensure that the entity-relation arrangement is valid.

III. Create a *Satisfy* Entity-Relation Problem
   a. Use the same entity-relationship construct from the previous problem.
   b. Submit the augmented "match" to the generic entity-relation problem solving system as a *Satisfy* problem type. This will propagate values to the A-E and AB-E entities so that the entity-relation problem is satisfied.

IV. Create a *Check Satisfaction* Entity-Relation Problem
   a. Use the same entity-relationship construct from the previous problem.
   b. Submit the augmented "match" to the generic entity-relation problem solving system as a *Check Satisfaction* problem type. This will check to see that all relationships of the A-E and AB-E sub-graphs are satisfied with the given "match" entity values and the A-E and AB-E entity values.
   c. If at least one augmented "match" is found to be satisfied, then continue, otherwise exit as failed.

V. Return the satisfying augmented "matches" to the user. If entity values have been identified as requested by the user (through ID relations in the A-E or AB-E sub-graphs) then display the values. If the entities are parametric, display the numeric value. If the entities are geometric, highlight the entities.

Table 3.2: The Design Exemplar Algorithm: Transformation (Summers. 2004)

I.	Execute the Validation Algorithm
a.	If Validation Algorithm has returned true, then continue, otherwise exit as failed.
II.	Create a *Validation* Entity-Relation Problem
a.	For each augmented valid "match" allow all entities to be free to be modified.
b.	For each "match" in the "working model" apply the associated relationships of the B-E, AB-E, and B-M sub-graphs.
c.	For each "match" in the "working model" create any entities found in the B-E, AB-E, and B-M sub-graphs. Also, create remaining relationships between newly created entities and other entities.
d.	Fix the values of the B-M entities so that they cannot be modified in this entity-relation problem.
e.	Submit the augmented "match" to the generic entity-relation problem solving system as a *Validation* problem type. This will check to ensure that the entity-relation arrangement is valid.
III.	Create a *Satisfy* Entity-Relation Problem
f.	Use the same entity-relationship construct from the previous problem.
g.	Submit the augmented "match" to the generic entity-relation problem solving system as a *Satisfy* problem type. This will propagate values to the B-E and AB-E entities so that the entity-relation problem is satisfied.
h.	Transfer the values of the results of the entity-relation problem to the design model.

To enhance the querying capabilities of the design exemplar system and to provide a way to re-use exiting design exemplars in authoring complex exemplars, the design exemplar system has been enhanced to support logical connectives [Divekar. et. al., 2004], vocabulary that may be used to compose complex exemplar queries, and static networks [Summers. et. al., 2004], authoring tools for reusing existing exemplars. These tools are discussed briefly in the following sections.

Logical Connectives

Logical connectives were introduced as a step towards developing the design exemplar system into a CAD query language [Divekar. et. al., 2003]. The existing design exemplar system supports AND, OR and NOT logical connectives. While, the logical connectives OR and NOT are explicitly implemented in [Divekar. et. al., 2004] and the AND logical connective is inherently supported by the exemplar system. The OR logical connective provides the flexibility of querying either one particular feature or another specific feature of a design model. For example, a design exemplar, as shown in Figure 3.9, can be authored using an OR logical connective to find a pair of lines in a model that are either parallel or perpendicular. When a model is queried using this exemplar, the pairs of lines that are either parallel or perpendicular to each other are found. As an illustration for the NOT block, Figure **3**.**10** shows a design

exemplar that is used to find a pair of lines that are perpendicular to each other but do not intersect at a point. Since, it is required that the lines *do not intersect*, the coincidence constraint is included into the NOT block of the exemplar.

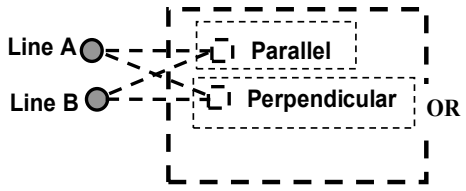


Figure 3.9:Exemplar to find a pair of lines that are either parallel or perpendicular

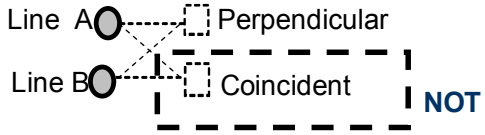


Figure 3.10: Exemplar to find a pair of lines perpendicular to each other but do not intersect.

Exemplar Networks

Anecdotal experience based upon several years of exemplar development across three different research labs (Arizona State University, Michigan Tech University, and Clemson University) suggests that design exemplar development is tedious and authoring exemplars that represent generalized characteristics becomes more tedious [Bettig. et. al.,1999] [Summers. 2004]. Research is being done to reduce the tediousness of building exemplars. Static exemplar networks [Summers. 2004] and similarity based exemplar retrieval [Anandan. et. al., 2006] accomplish this task in two different ways. While the former aids in building new exemplars by allowing the user to reuse existing exemplars, the latter helps the user in selecting the exemplars that can be reused.

The static exemplar network was developed to overcome the limitations of the design exemplar with respect to its reusability. The static exemplar network reuses exemplars by integrating them into other exemplars to accommodate the inclusion of new characteristics. This would reduce the overhead of authoring exemplars from entity level. Four basic situations where the design exemplars can be reused in a network are presented with examples below identified as combining characteristics, relating characteristics, refining and generalizing characteristics, and combining tasks [Summers. 2004]. For example, exemplars to find a boss and a solid body can be networked to find a solid body with a boss (Combining characteristics). Further, a new distance constraint between the axial line of the boss and the walls of the solid body can be introduced into the resulting exemplar (relating characteristics). Also, the type of curve bounding the cylinder of the boss can be set to a specific type (refining) or a generic curve (generalizing characteristics). Figure 3.11 shows a static network, where various nodes, each holding an exemplar are linked together to form a new exemplar.

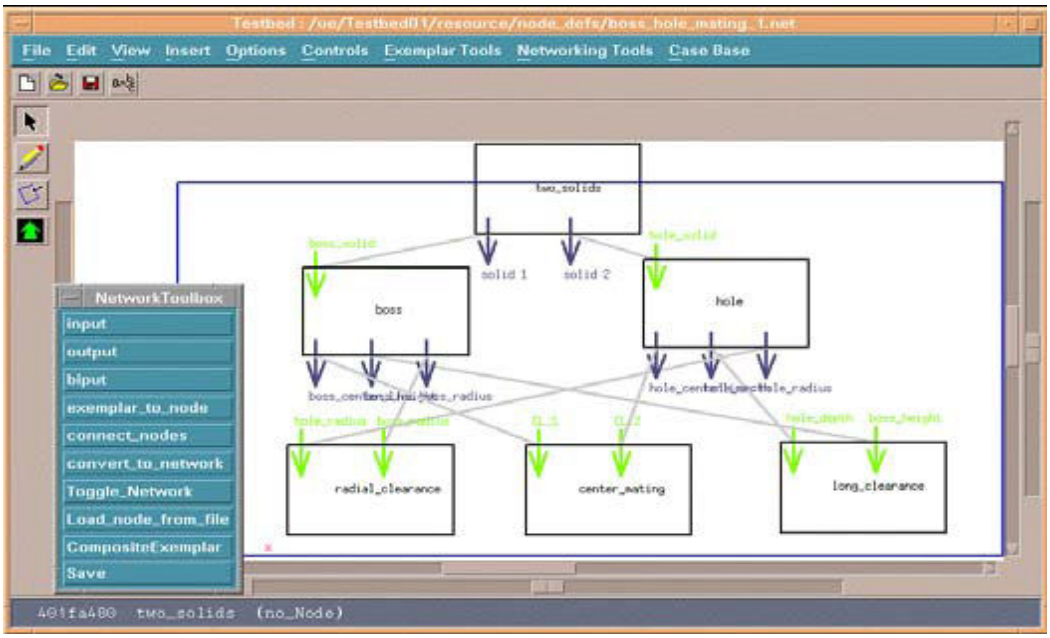


Figure 3.11: Figure Figure showing a Static Network [Summers. 2004]

The exemplar network has also been suggested to provide a level of abstraction between the CAD developer and the finite details of the design exemplar's bi-partite entity-relation graphs. Hence, the static exemplar networks provide a way to compose complex exemplars by reusing existing exemplars. However, this approach does not provide a way for iterative re-use of exemplars.

Summary

This chapter has introduced the concept of a design exemplar. The various capabilities of the design exemplar system proposed in the literature have been summarized and presented with examples. Tools like the logical connectives and the static exemplar networks that were provided to support complex querying capabilities and re-use of exemplars for authoring complex exemplars have been discussed.

CHAPTER 4

THE DESIGN EXEMPLAR AS A VISUAL PROGRAMMING LANGUAGE

The simplicity of programming and the ease of visualization achieved due to the elimination of textual constructs and syntax for programming, which varies from one programming language to the other, forms the rationale for considering visual programming for this research. The essential components of a visual programming language, icons, iconic system and a visual language compiler, were identified and presented in Chapter 2. Further, the design exemplar was presented in Chapter 3 as a data structure used to represent design data, which can be used to perform tasks like querying information, design validation, pattern matching and model modification. Topologic, geometric and parametric entities and constraints found in a design model form the basic components for design exemplar authoring. This chapter explores the potential of using the design exemplar as a visual programming language for mechanical engineering design automation tool development.

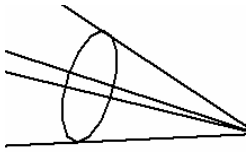
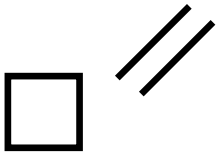
This chapter seeks to identify the components of a visual programming language that are missing from the existing design exemplar system. A comparison is drawn between the various components of the design exemplar system and a visual programming language to identify the essential components that need to be introduced into the design exemplar system to extend it into a visual programming language. Since, the complete development of a visual programming language remains out of scope for this thesis, the components that can be addressed here are identified and the methodologies employed for developing these components is presented. Finally, the components that are out of scope for this thesis are identified and left for future work.

The following sections present a comparison of the three important components of a visual programming language (The Icons, The Iconic System and The Compiler) which have been identified in Chapter 2, with the components of the design exemplar system. Here, the design exemplar system is also checked for its capability to support important programming constructs like, looping and conditional branching, identified in Chapter 2

The Design Exemplar – Icons

Though the design exemplar was introduced for extracting and querying design data [Bettig. et. al., 1999], a closer observation reveals that this representation of the design exemplar is analogous to the use of icons for representing visual sentences. Specifically, the topologic, geometric, and parametric entities and relations used for representing design data in a design exemplar are analogous to object and process icons [Chang. 1990] used for visual sentence authoring in an iconic visual programming language. The Merriam-Webster dictionary [Merriam-Webster. 2007] defines the word icon as a word or graphic symbol, whose form suggests its meaning. In the context of a visual language, an icon is an object that visually represents a meaningful programming construct. An icon has two components: the physical part and the graphical part [Chang. 1990]. While the image, picture, or figure that is used for representation is the physical part of the icon, the information useful for processing it is the logical part. The two basic criteria that characterize the icons in a visual programming language, the visual information and the logical information, are believed to be existent in the representations used for entities and relations used for design exemplar authoring. As stated in Chapter 3, the design exemplar system supports visual interaction for composing exemplars using a clearly defined set of entities and relations. Also, these representations, in addition to providing a visualization of the geometry or topology associated with it hold the necessary information of the entity that is required for processing. For example, the symbols used in the exemplar system to represent a conical surface, geometric entity and a parallel constraint, a geometric relation are shown in Table 4.1.

Table 4.1: Figure showing some Icons used in the Design Exemplar System and the Visual and Logical Information in them

Icon	Visualization for	Logical Information
	conical surface	• Information relating to the face geometry • Position • Radius • Angle of the cone • Bounds
	parallel relation between two features	• Information related to the constrained elements • Information related to the conditions for the parallel constraint.

The representation used for a conical surface as shown in Table 4.1 provides the user with a visualization of conical surface and also holds the information like the position of the center, the radius of the base circle and the angle of cone that are required for processing. Similarly the representation for a parallel relation holds the information related to the two entities that this relation is used to constrain. Hence, it can be stated that the visual objects used to represent the entities and relations in the exemplar system are analogous to the icons of an iconic visual programming language. It may be noted that the iconic languages defined so far have dealt mainly with arithmetic or logical processing. While the design exemplar is presented as a tool for geometric data processing, this research focuses on developing it into an iconic programming language.

Icons used to represent the basic geometric entities and relations in the design exemplar system can be categorized as the elementary icons of the design exemplar system,[Chang. 1990]. These icons can further be separated into object and process icons. The icons that represent geometric, topologic, or parametric entities and relations can be grouped as object icons of the design exemplar system. A list of object icons supported by the design exemplar system can be seen in Table 4.2 and Table 4.3 respectively. The graphical part of the icon is shown to the right of these tables while the entity /relation (geometric, parametric or topologic) it is representing is shown to the left. It should be noted that these tables do not present an exhaustive list of the icons supported by the exemplar system. Since, the design exemplar system also allows the inclusion of new icons; more icons are expected to be developed to address the new needs. This was demonstrated in [Summers. 1981].

Table 4.2: Figure showing a list of Object Icons in the Design Exemplar Vocabulary

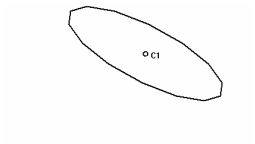
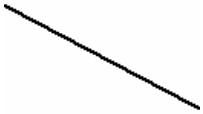
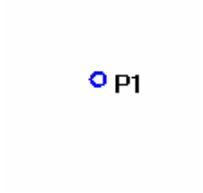
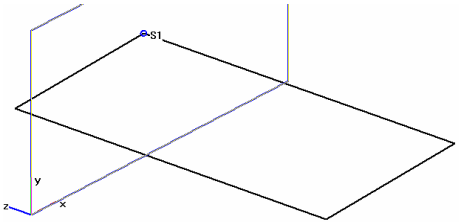
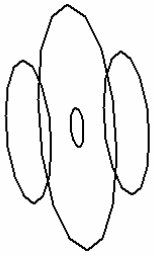
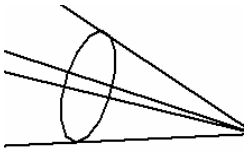
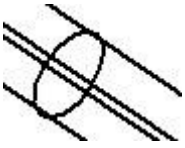
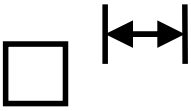
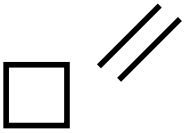
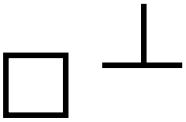
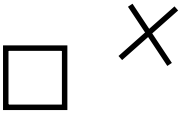
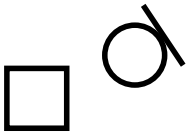
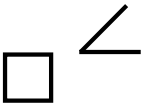
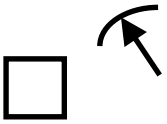
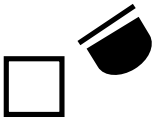
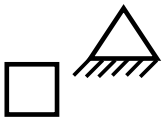
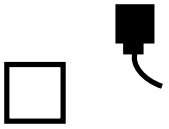
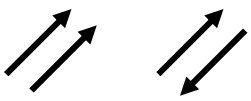
NAME	ICON
Circle	
Straight Line	
Point	
Plane	
Torroidal surface	
Conical surface	
Cylindrical surface	

Table 4.3: Object icons (representing geometric relations) used in the Design Exemplar System

NAME	ICON
Distance constraint	
Parallel constraint	
Perpendicular constraint	
Coincidence constraint	
Tangent constraint	
Angle constraint	
Radius	
Boundary	
Fixed constraint	
Id constraint	
Same direction / opposite direction	

The design exemplar system also supports the use of complex icons [Chang. 1990] composed by using other object and process icons. The nodes of a static exemplar network can be considered an example of a complex icon [Summers. 2004]. Here, the design exemplar is enclosed in an enclosed in a rectangle, while the input and output ports connected to the appropriate entities and relations of the exemplar can be connected to other nodes (Figure 3.11). This node intentionally hides most of the entities and relations of the exemplar and exposes only the ones that will be connected to other nodes. This complex icon, the node icon, is combined with the other node icons using process icons called ports and connectors to form a visual sentence, or an exemplar network. These networks are then compiled into a new design exemplar that incorporates all the elemental exemplars linked together in the network. Analogous to the functions in a functional programming language [Paoluzzi. 1995], complex icons can be used both as visual sentences or icons in a visual sentence. For a detailed explanation of the static exemplar networks the reader is referred to [Summers. 2004]. It should be noted that the design exemplar composed using various other object and process icons can itself be viewed as a complex icon. For example, a design exemplar representing a line segment formed by a line entity and bounded by two points, as shown in Figure 4.1, can be viewed as a complex icon representing a line segment.

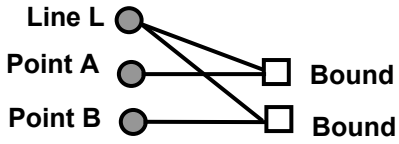


Figure 4.1: Complex Icon representing a line segment

The Design Exemplar - Iconic System

The iconic system of an iconic visual programming language consists of a structured set of icons and the associated iconic operators composed into a syntactically accepsentence. In this section, the iconic system of the design exemplar system is presented based on formal specifications of iconic systems [Chang. 1990]. The design exemplar system has a structured set of iconic operators that can be used for a visual sentence or complex icon authoring. Since, the object icons of the design exemplar system represent entities and relations that are used to describe geometry, it can be said that the operator icons operate on these entities and relations to relate them. For example, in Figure 4.1, the bounding relation between the

line L and the points A and B is shown by an arc joining the entities and the relation icon. These arcs are iconic operators that represent an explicit relation. Hence, Figure 4.1 can be read as: *Line L is explicitly bound by point A, Line L is explicitly bound by point B*. The list of iconic operators supported by the design exemplar system is shown in Table 4.4.

Table 4.4: Iconic Operators of the Design Exemplar System

NAME	ICON
Match (explicit)	
Extract (Implicit)	
NOT block	 NOT
OR block	 OR

The NOT, OR blocks are operators, proposed to enhance the querying capabilities of the design exemplar system. While the OR block may be used in situations where it is required to combine two or more object icons, to represent a combination of features or conditions, the NOT block is used to group icons that represent features or conditions that are expected not to be present in the model. While the icons presented in Table 4.4 forms the first type of operator icons, the exemplar system also supports another type of operator icons are used to distinguish between alpha, alpha_beta and beta states of a model. These operators operate on both the object icons and iconic operators and are graphically represented by a change in the color of the object icons and the iconic operators. The logical and physical information in the icons and operators is shown in Table 4.5.

Table 4.5: Physical and logical parts of the various icons in the Design Exemplar System

Icons	Physical Part holds	Logical part holds
Object Icons *Eg: Circle*	An image Figure showing a circle of variable radius and whose center remains fixed at the position it is inserted	Information relating to the topology, center, radius
Complex Icons: *Eg: Node in a Static Network*	A rectangular block named after the exemplar included in it	Hold a design exemplar and information regarding the direction of control flow.
Icons Operators *Eg: dashed line segment.*	An image Figure showing a dashed line	Information that suggests that the constraint is implicit

Since, the icons and the iconic operators are found in the existing in the design exemplar system. The design exemplar may be defined as

"A visual sentence useful for representing, investigating or modifying a geometric or parametric design model, composed by appropriately combining object icons (entities and relations) using iconic operators"

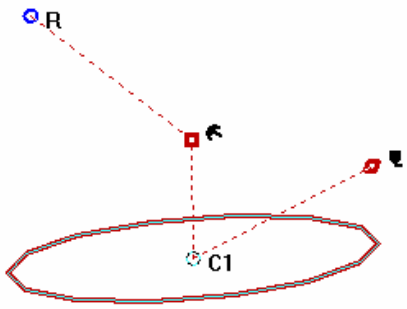


Figure 4.2: Visual representation of a Circle with Radius R

Figure 4.2 shows an exemplar representation of a circle bound by a radius. This representation, as can be seen from the figure can be called a visual sentence used for investigating the radius of the circle. Different object icons are used for representing the Circle, Radius Parameter, Radius constraint and the ID relation. Iconic operators for implicit relations are used to relate these icons.

Further, the design exemplar may be classified into the connection based visual programming language as suggested by Costagliola [Costagliola. et. al., 2002], where a visual sentence is formed by

connecting the visual objects of the exemplar vocabulary. Further, the design exemplar uses, overlapping type of connections, where the connections are visualized by overlapping attaching points or regions, for the visual sentence formation [Costagliola. et. al., 2002]. Figure 4.2 shows an exemplar (visual sentence) formed using the primary object icon for the *Circle* (geometric entity) and process icons for *Radius*, *Id Constraint* and the *Extract Constraint*. Since these icons overlap each other to form a visual sentence, this exemplar may be presented as an example for a visual sentence where the connections are of overlapping type.

The Compiler

As high level programming languages are primarily developed to improve usability, it is required to translate these programs into a lower level language for processing. This translation of a program written in a higher level language to a lower level language is done by the compiler. Hence, the compiler is seen as an important component in a programming language. It is a program that accepts high level description of a desired system of a program and converts it into a computer executable. The compiler of an iconic visual programming language accepts formal specifications of an icon-oriented system interactively, and generates a realized icon-oriented system as its output [Chang. 1990]. This chapter tries to establish the design exemplar as a visual programming language that can be used to develop applications for geometric processing. Hence, the compiler associated with it is a mechanical compiler and as suggested by Ward [Ward. et. al., 2003], is expected to:

- Provide a high-level and reasonably flexible language for users to accurately define the desired mechanical design interactively, using the set of uniquely defined set of icons and operators.
- Accept any syntactically correct and semantically meaningful input provided the form of icons.
- Interpret and convert this iconic input into realizable execuoutput.

The ability of the design exemplar system to represent mechanical design using a set of icons and operators has been presented in the previous section of this chapter. The following sections discuss the syntactic and semantic analysis and icon interpreter associated with the compiling system.

Syntactic and Semantic Analysis

As discussed in Chapter 2, a syntactic and semantic analysis is performed on the visual sentence before converting the source code (here, visual sentence) into an object code. This analysis is performed based on the defined grammar associated with the design exemplar creation. Since, the exemplar system

deals with geometric models, the rules governing the construction of the geometry form the basis for syntactic and semantic analysis. Meaning that, only the design exemplars that can be represented geometrically can be composed. Compatibility of geometric entities and relations is tested before a constraint/relation is imposed on an entity of the model and the creation of a relation between incompatible entities and relations is not permitted. For example, imposing a radius constraint on a straight line is not possible in geometry construction, hence this type of iconic relations are not possible to construct in the design exemplar system.

Since this analysis is performed while the design exemplar is being authored; the visual sentence formed at the end is both syntactically correct and semantically a valid design model. The ability of the design exemplar system to perform this analysis while composing comes to it by the virtue of the information stored in the logical part of each icon. When the user tries to impose a relation on a geometric entity, the logical information in these icons is checked for compatibility and the possibility of a relation is decided. For example, when a radius relation is imposed on a line, the logical information in the radius icon and the line icons is checked. The logical information in each of these icons is shown in Table 4.6.

Table 4.6: Logical information in radius relation and straight line icons

Logical Information in the Radius icon (Relation)	Logical information in the Straight Line (Entity)
• The radius value (a numeric value) • The edge geometry (of the entity this relation is applied on)	• The edge geometry of the entity • Direction (magnitude and direction) • Cartesian point

When the radius relation is imposed on the straight line, the edge geometry that it is associated is a straight line. This information is found in the logical part of the straight line icons. This edge geometry is checked against the edge geometry information inside the logical part of the radius relation and if compatible, the relation is formed, else the relation is not formed. However, the edge geometry information inside the radius icon suggests that this relation can be associated only with entities that have a circle in them, like circles, cylinders, cones etc. Hence, when the check is performed on the line entity the relation is not formed because the edge geometry of the line entity does not match with any of the edge geometries that are compatible for the radius relation. When the radius relation is applied on a circle entity, the relation is valid since the edge geometry information found in the circle entity matches with the edge geometry information of the radius relations.

This section presents the ability of the design exemplar to perform a syntactic and semantic analysis while authoring the visual sentence. Hence, this aspect of the compiler required for a visual programming language already exists in the current exemplar representation. The following section presents the next important part of a compiler, the icon interpreter.

Icon Interpreter

The icon interpreter is viewed as the most important component of an iconic visual language compiler. It accepts a formal icon-oriented user input constructed using the icons, iconic operators and the iconic system and generates a realizable output. The design exemplar, initially proposed as a data-structure for retrieving geometric information is not supported by compiler. The geometric processing associated with pattern matching, query extraction and model modification are performed by the constraint solvers embedded externally into the design exemplar system. In order to achieve the full functionality of a visual programming language, the design exemplar system should be able to compile the user input visual statements into executables. Though the complete development of such a compiler remains out of scope for this research, a possible system design for such a compiler is presented in Figure 4.3 based on the SIL-ICON compiler proposed in [Chang. 1999].

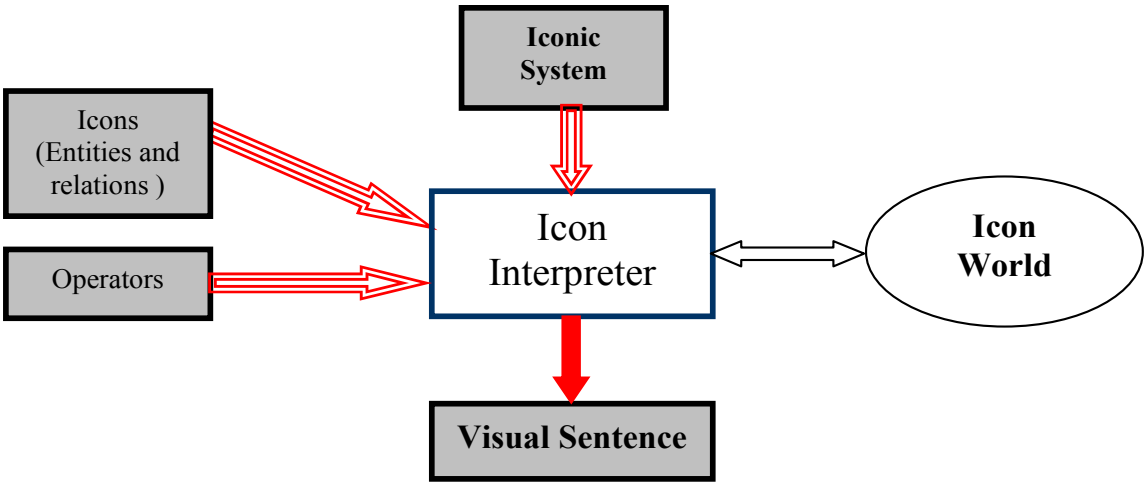


Figure 4.3: The System Diagram of the Design Exemplar Compiler extended from the SIL-ICON Compiler [Chang et. al., 1989]

The icons and the iconic operators of syntactically and semantically combined using the grammar associated with the iconic system form a formal specification of an icon-oriented system which is transformed into a realizable icon-oriented system (the visual sentence) by the icon interpreter. The icon

world is an infinite set of icons and any icon set selected by the user for a visual sentence creation is a subset of the icon world. The development of a compiler to fully support visual programming in the design exemplar system remains out of scope for the project and hence is not addressed in this thesis.

This chapter has sought to identify the components of a visual programming language that are missing in the design exemplar system. The representations for entities and relations combined with the design exemplar grammar were found adequate to categorized into a fully developed iconic system. The syntactic combination of icons, governed by the solid modeling kernel, ACIS, provides for the creation of geometrically valid designs. In addition, it should be noted that a geometric exemplar representing a design provides a unique definition its components. These two aspects were identified to be very important requirements of a design language [Poauluzzi. et. al., 1995]. The other aspect, design encoding, identified by Poauluzzi, suggests that user should be concerned only with the semantic meaning of the source script and not with the development of an efficient code. However optimizing the source code remains a very important aspect for the efficient representation of the design, and is usually done as a part of the compiling process. Though the exemplar representation provides for easy development of the source code, in the form of icons representing entities and constraints, it does not provide for compiling it. So, the development of a compiler to support design encoding in the design exemplar visual programming language is seen to be very important. However, the development of the same is too huge a task and remains out of scope for this research.

Apart from the basic components that are necessary for any programming language, it is also required by these programming languages to support certain functionalities that facilitate easy programming. In the following section, these aspects or capabilities of a programming language are identified and the possibility of the design exemplar boasting these capabilities is checked.

Other Aspects of Programming Language

The need for the simplifying the process of programming and ease of data visualization formed the rationale behind evolution of programming languages from low level to high level. Methods to support the easy handling of data have been developed to support the user. For example, arrays, data structures, conditional statements, looping, conditional branching etc are supported by the current day programming languages. Though it is not necessary for programming languages to support such methods, current day computing requires easier methods of data handling.

Data Structures

Data structures are used in textual programming languages to hold together the information related to an object. These objects are directly used for programming. It should be noted that the icons of the design exemplar system are similar to the data structures supported by textual languages like c, c++ etc. The information required for processing geometric entities or relations is present in the logical part of their icons. Complex icons can further be defined as data structures a higher level of abstraction. Hence, it can be said that the design exemplar system supports data structures that hold geometric, topologic and topologic information related to an object. Arrays on the other hand are data structures that hold a group of homogeneous data together, making it easy to access and manipulate. For example, the numerical values in an (N X 2) matrix can be stored in the form of a two-dimensional array and element of the matrix can be accessed directly by the appropriate index values. However, the design exemplar system, as proposed here uses geometric entities and relations for programming or creating visual sentences that represent meaningful geometric objects. It should be understood that merely grouping similar entities or relations does not produce meaningful design objects. Hence, the need for an array type of data structure in the design exemplar representation is felt irrelevant.

Conditional Branching

Conditional branching allows the execution of a set of statements only when a certain condition is met. These conditions are called the conditional statements. The conditional statements are requests to the computer to make the choice of execution based on a condition. For example, in higher level programming languages like C and C++, when the computer finds an “IF” statement, the condition following it is evaluated. If this condition returns true, in other words, if the condition evaluates to a non-zero value then the statements listed below THEN are executed, other wise the control is moved to another part of the program.

The working of the design exemplar algorithm is similar to evaluating a conditional statement, in that, when a model is queried against an exemplar, the graph representing the model is checked to find a match for the exemplar graph pattern. This check returns a true when a match is found and returns a false if the match is not found. Since this is analogous to the evaluation of a conditional statement in a textual programming language, it can be said that the design exemplar system supports conditional statements. However, it should be noted that after the check is performed and a match found, the design exemplar

cannot operate on the model any further. On the other hand, conditional branching is achieved only when an operation is performed on the data/model based on the result of the check is performed. Hence, it can be said that the existing design exemplar system does not support conditional branching.

In conditional branching, as explained before, the system performs a series of operations after performing a check. This essentially means the condition and the operation are sequentially executed. However, the design exemplar as presented in Chapter 3 is a declarative representation; hence it does not support sequential operations.

Looping

A sequence of operations executed several times in succession until a termination condition is satisfied is called looping. Most of the fourth generation languages support two types of loops: count controlled and condition controlled. In a count controlled loop, a set of operations are repeated a certain number of times and in a condition controlled loop, the loop is executed either while a termination condition is satisfied or while the condition is not satisfied. For example, Table 4.7 shows a simple loop in c++ where a "FOR" type of count controlled loop and a "WHILE" type of condition controlled loop is used to increment the value of 'j'.

Table 4.7: Figure showing – Count Controlled and Condition Controlled loops

```
void main()
        {
        int i;
        int j=0
        for (i=0;i<5;i++)
                {
                j++;
                cout<<endl<<"The value of j is :"<<j;
                }
        }
```

```
void main()
        {
        int j=0;
        while(j<5)
                {
                j++;
                cout<<endl<<"The value of j is :"<<j;
                }
        }
```

The FOR loop shown here is executed as long as the value of the counter variable, 'i', remains less than five. However, the value of 'i' is being incremented each time the loop is executed. Hence the loop terminates when the value of 'i' is equal to 5, that is, after five executions of the statements enclosed in the loop. The execution of the WHILE loop shown in right column of the is similar to the FOR loop but here, no counters are used and the condition is set on the variable itself.

Looping eliminates the need for repeatedly writing a set of statements, which some times can be very tedious. Also, in situations where the number of iterations required is not known a priori, looping

proves to be very useful. Hence, it is seen as a very important of the current day programming languages. However, the declarative nature of the design exemplar limits it from supporting conditional branching.

Problem Identification

Since, the objective of this research is to establish the design exemplar as a visual programming for developing mechanical design applications, a thorough study of the various aspects of programming language is performed and presented in the previous sections. This study has suggested that, high-level programming languages existing today support constructs and routines that enable easy handling of complex data. A few important data handling constructs commonly supported by high-level programming languages were identified and the ability of the existing exemplar system to support them was checked. It is found the design exemplar system, being a declarative representation, does not support data handling methods like looping and conditional branching. These methods can be useful for geometric processing in, where a set of operation are required to be performed.

To explain the usefulness of these constructs in the envisioned visual programming language that can be used for developing geometric processing, applications, an example involving the design of a gear train is presented here. Given the maximum speed ratio between the input and the output gear of the gear train as *n:1*, it is required to design a gear train such that the condition for maximum permissible gear ratio to prevent undercutting in a pair of gears, given as 6:1 [Bhandari. 1997], is obeyed. The method used to achieve this design task is called the "Dividing Method" and is out-lined below:

1. A gear pair with the given ratio is considered. (A gear pair with n: 1 gear ratio is considered for this purpose)
2. If the maximum permissible gear-set ratio (It is considered to be 6:1 for this problem) is less than the ratio on the first gear pair (n : 1), a gear pair with the maximum permissible gear set ratio is introduced and the ratio of the first gear pair is set to n:1 (where n= n/6), thus maintaining the given input to output ratio of n: 1.
3. Step 2 is repeated until the ratio of the first gear pair (n:1) is less than the maximum permissible gear ratio (6:1).

The algorithm describing each step that needs to be performed to achieve this design task is shown in Table 4.8.

Table 4.8: Algorithm for a Gear Train Design (Given, the ratio)

1. Introduce a pair of gears (Input Gear Pair); proceed to step 2.
2. Set the ratio of the gears to n:1 and identify the output gear; proceed to step 3.
3. Check the ratio of the last gear pair (will be called as Gear Pair- N); if it exceeds the maximum permissible gear ratio (6:1), introduce a new pair of gears into the model and proceed to step 4; else, exit.
4. Identify the last two gear pairs (Gear Pair-N and the newly added gear pair). Set the ratio of Gear Pair- N to 6:1and the ratio of the newly introduced gear pair to (n/6): 1; (n=n/6) proceed to step 5.
5. Identify the last two gear pairs and set the distance between the shafts for Gear Pair- equal to the sum of the pitch circle radii of its two gears ; Set the distance between the shafts for the newly introduced gear pair (in step 4) equal to the sum of the pitch circle radii of its two gears; proceed to step 6.
6. Repeat step 3.

Designing a gear train in this method requires performing operations defined in steps 1-6 (in the algorithm) in a sequential fashion, in that order. Also, it can be seen that operations in steps 3-6 are repeated until the check in step 3 is not satisfied. Hence, this design task requires:

1. Sequential processing of geometric information
2. Conditional processing of geometric data (Performed at step 3)
3. Repeatedly perform a set of operations (Steps 3-6).

To enable the handling of such problems, the existing design exemplar system should be enhanced to support sequential processing of data, while providing constructs for conditional and iterative processing of design data. In other words, the envisioned design exemplar visual programming language should be able to process design information in a procedural fashion.

A system that uses icons to represent system information pertaining to dynamic systems, in a sequential fashion is Simulink. It is a software used to model, simulate and analyze dynamic systems. To explain this idea, the representation and processing of a simple spring-mass system in Simulink is presented

is presented as an example. Simulink has a well defined block library of sinks, sources, linear and nonlinear components, and connectors that may be used for modeling, analyzing and simulating dynamic systems [Simulink. 2004]. These building blocks can be assembled in a GUI (Graphic User Interface), using simple click-and-drag operations.

The second order differential equation '*a*' governing the relation between the acceleration and displacement of the mass for the spring mass system shown in Figure 4.4 may be appropriately represented and solved using Simulink.

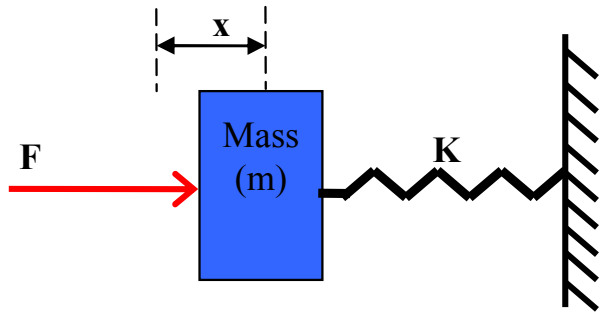


Figure 4.4:A spring -- mass system acted upon by a force F

Figure 4.5 shows the Simulink model representing equation *a.* Blocks representing gains, integrators, input type, and output visualization are connected to represent equation a. The variation in displacement of the Mass (m) with time under different force inputs can be simulated.

$$1/m(F - Kx) = \ddot{x}$$ -- Equation (a)

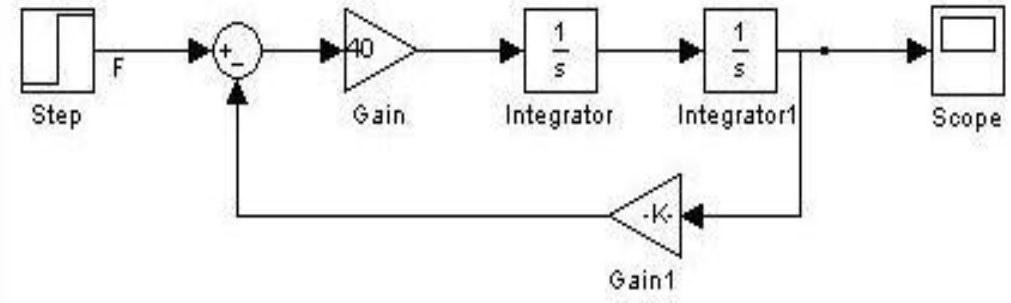


Figure 4.5: Simulink Model representing Equation (a)

Though procedural methods have been previously employed for processing algebraic and logical information, the validity of such an approach in a visual programming language for geometric applications like the design exemplar system remains a question to be investigated. Further, the new method poses new questions about the complexity of processing and the size of the solution set. The questions addressed in this research and the hypotheses are listed below. The following chapter presents an algorithm for the envisioned procedural approach called the "Dynamic Networking" approach and presents case studies to validate the hypothesis made.

Q 2: Can a procedural approach be employed in a visual programming language for mechanical design?

Q 2.1: Can a procedural approach be developed to extend the exemplar to support conditional branching?

Q 2.2: Can a procedural approach be developed to extend the exemplar to support looping?

Q 2.3: How does the new approach affect the complexity of solving?

Q 2.4: How does the new approach affect the number of solutions obtained**?**

CHAPTER 5

DYNAMIC NETWORKING OF EXEMPLARS

The existing design exemplar system does not provide for automatically performing a series of operations on a given design model. Such processing requires listing the exemplars, each representing an operation to be performed and subjecting the model to each one of them in a sequential fashion. This is analogous to the execution of statements and procedures in a procedural programming language. The existing design exemplar system does not provide for such processing, instead such tasks are handled manually, where the user picks the exemplar representing a desired operation and then the model is subjected to it. This approach is rather time consuming and can be tedious when the number of operations that need to be performed is high. On the other hand, in a visual language perspective, programming constructs like looping and conditional branching, which are believed to ease the task of programming by providing a way to handle large sets of operations can be implemented only in an environment that provides for sequential processing of data. Hence, enhancing the design exemplar system to provide a way for automating sequential processing of design information would prove to be a useful.

To address these issues and to improve the usability of the design exemplar system for solving design problems, the concept of "Dynamic Networks" is presented in this chapter. The dynamic networking approach is presented as a way to achieve procedural processing of design data while making use of the declarative design exemplars for this purpose. The application or the use of design exemplars to process design data in this approach is analogous to the use of procedures in a procedural programming language. It should be noted that the working and purpose of these networks (dynamic networks) is different form the static networks proposed in [Summers. 2004]. While the dynamic networks provide for sequentially solving the constraint problems representing each exemplar of the network separately, the static networks combine them into a single constraint problem. The dynamic network, apart from enabling the reuse of exemplars, which is the most important advantage of the static exemplar networks, also provides for iterative use of exemplars without making multiple copies of them.

There are three important components in a dynamic network: (a) Nodes, (b) Output Ports and (c) Connectors. A dynamic network is formed by connecting a series of nodes in a logical sequence thorough

the ports, using connectors. While the ports and connectors serve the purpose of logically connecting the nodes and representing the flow path or the order of execution, it is at the nodes that the processing of design data takes place. Each node of the dynamic network holds a design exemplar that is used to process a design model at that node. The control propagation at each node of the network depends on the result of the check performed or choice made at it and a series of exemplar checks is performed through the network before a final result is obtained. The use of this approach for part/system design can be see as analogous to a production system, where each rule is represented using a design exemplar and an appropriate rule is fired based upon the result of the check performed using the immediately preceding rule (here, the design exemplar) such that an appropriate rule is fired based on the result of the immediately preceding rule. For example, the specifications of a ball bearing model identified from a database may be modified based on a set of rules using a system such as the one proposed here. As this approach is rule based, the design developed using this approach is less error prone and easily obtained.

Dynamic Node

The dynamic node is introduced as a data-structure to hold an exemplar required for processing design data. It, in addition to being a point in a network where an exemplar check is performed, also provides an option for choosing the modification technique and the subsequent direction of control propagation within the network. The dynamic node proposed here is different form the static node proposed in [Summers. 2004], in that it has a broader function than merely hiding exemplar data. The exemplar node in a static network was proposed as a simple data structure developed to hold a design exemplar for processing design data while hiding the exemplar data from the user. For convenience, the dynamic node will hereafter be referred to as "Node".

Based on the tasks performed at the various levels, the dynamic node is divided into the exemplar and the decision blocks. The Exemplar block, which consists of the exemplar and the node name, performs the exemplar check and returns a true or false. The decision block, which is internally connected to the exemplar block within the node, provides an option for changing either one or all of the matches found as a result of this check. The node name can be used to distinguish between the nodes of the network and indicate the operation performed at the node or the exemplar in it. While, the exemplar block of the node performs the match functionality of the design exemplar in it, the decision block provides a way for the modification. Each node has four output ports: *True, TrueOne, TrueAll* and *False*, associated with it.

These ports facilitate connecting a node with the other nodes of the network.

Figure 5.1 shows the internal structure of a dynamic node with each section represented in a different color.

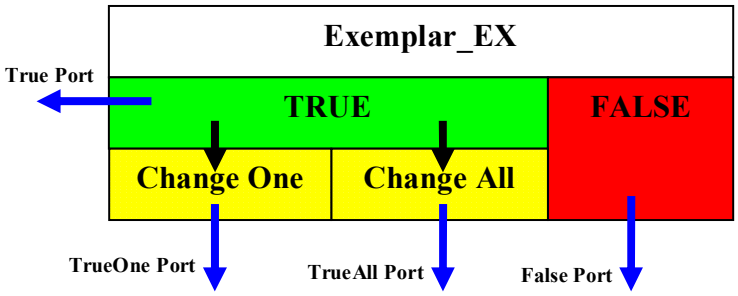


Figure 5.1: Complete Exemplar Node for Dynamic Networking

Though the node does not have to take the name of its exemplar, in this case *Exemplar_EX*, it will in this thesis for convenience. The *Change One* and *Change All* boxes shown in yellow, form the Decision Block of the node, while the remaining aspects, the *True*, *False*, and the *Exemplar_EX* boxes form the exemplar block. When the result of querying a model against the exemplar in the node is false, meaning that a match is not found, the next exemplar node in the network connected to this part (the *false* port) of the node is fired. On the other hand, if the check returns true, three different network paths may be followed. One is the direct external path through the output *true* port that can be used when the operations intended to be performed at the node do not involve modifying the design model, while the other two are through the decision block that is internally connected to the *True* part of the exemplar block. These ports may be used when the type of processing at this node involves modifying the matches found after the exemplar check and depending on the choice of the modification method they are named as *True One* and *TrueAll* ports. The *TrueOne* port of a node is used to connect it to the next node of the network when one of the matches found after the exemplar check is to be modified. On the other hand, the *TrueAll* port is used for this purpose when all of the matches found are to be modified. The nodes in the network are linked using lines called "Connectors". A connector joins the output port of one node with the exemplar block of the other. The ports in a node and the connectors in a network primarily serve the purpose of representing the connections. For convenience of representation, the dynamic nodes will hereafter be represented as show in Figure 5.2. In this representation, unless otherwise stated, the modification of matches is performed one at a time.

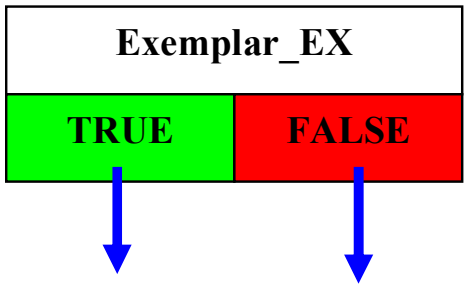


Figure 5.2: Simplified Exemplar Node for Dynamic Networking

Based on the re-use of nodes, the dynamic networks that may be formed are identified as pure and cyclic networks. A detailed description of these networks is presented in the following sections.

Pure Network

A pure network is formed by connecting a series of nodes in a sequence, such that the design data is processed not more than once at each node. Each model is evaluated using the exemplar at each node and the control is branched into the subsequent nodes based on the result this check. Hence, a series of constraint problems, each representing the exemplar in a node are evaluated sequentially with the sequence of evaluation being dynamically decided at each node. In the case of a pure network, none of these constraint problems are evaluated more than once. It should be noted that the dynamic networks also support the use of exemplars created by combining two or more exemplars with logical connectives, referred to here as the hybrid exemplars. In this case, each node is represented by a set of constraint problems. In a visual language perspective, the pure networks, while providing a procedural representation of the operations using declarative design exemplars, also provide for conditional branching.

Sub-networks representing varying characteristics of a design model may be formed at various levels of the network to collectively represent the various design features that may arise while querying the model. Figure 5.3 shows a hypothetical pure network used for illustrating the concept of sub-networks. The pure network in Figure 5.3 has sub-networks at three levels, each demarcated by a different type of dashed rectangle. The nodes Exemplar_1, Exemplar_2, Exemplar_3 form the first level of the network, while Ex_1_1, Ex_1_2 form a sub-network at the second level and Ex_1_1_1, Ex_1_1_2, Ex_1_1_3 form a third level sub-network.

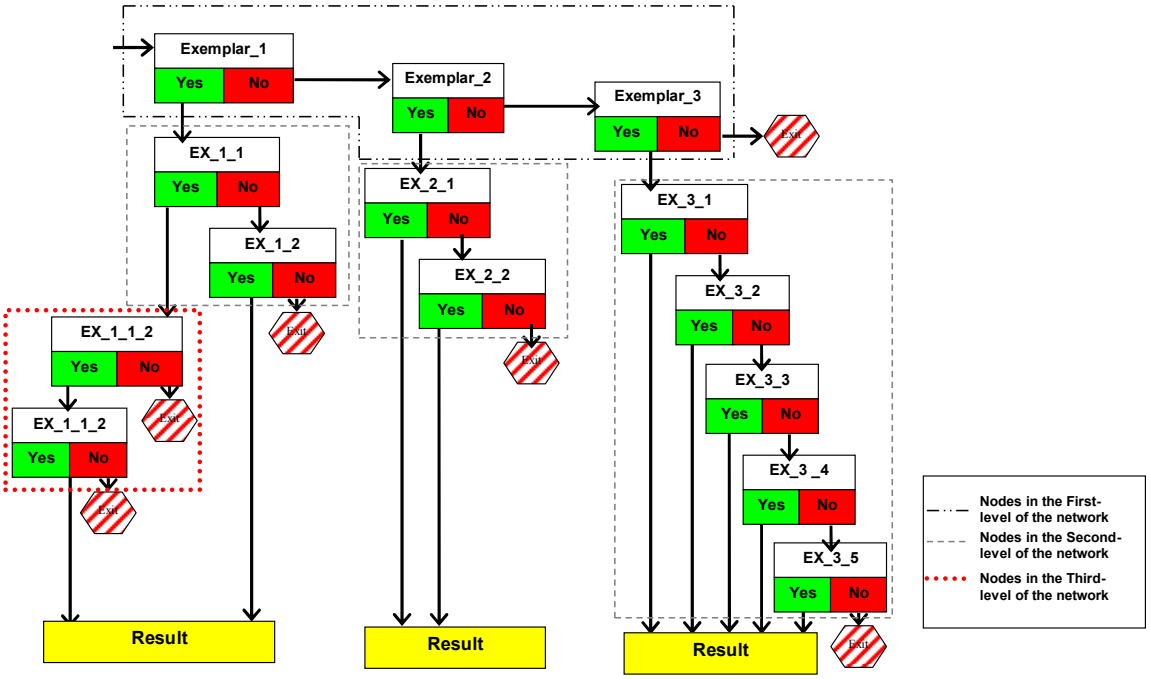


Figure 5.3: A Pure Network Figure showing its various levels

In the Figure 5.3, a design model is queried sequentially against each exemplar in one level of the network. At the node that returns true, the control branches into the next level to perform subsequent checks. This process of checking against exemplars in a level and branching off into a next level network when one check returns true continues until a final result is obtained.

Cyclic Network

Dynamic networks that involve the use of one or more nodes repetitively in an iterative fashion are called Cyclic Networks. While the control propagation from one node to the other remains similar to pure networks, these networks facilitate the use of a set of nodes iteratively until a desired condition is satisfied. The working of a cyclic network is analogous to looping in a programming language, where a set of logical operations are preformed until a condition is satisfied. These networks can be used for situations involving identification, modification, elimination or inclusion of more than one similar feature in a design model. It should be noted that these networks provide for re-using an exemplar without making copies of it.

Cyclic networks can also be used in design problems that require identifying features with maximum or minimum parameters. For example, to find the boss with the smallest or highest radius value in a model shown in Figure 5.4 and change its value to, say 20 mm.

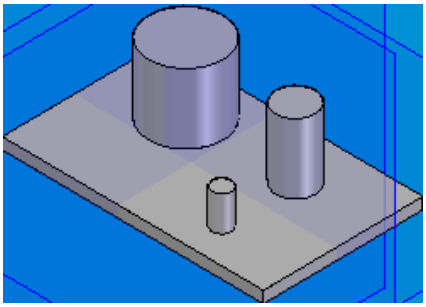


Figure 5.4: Model with Three Bosses

To modify the radius of the smallest boss in this design problem, it is first required to identify the boss with the smallest radius value. This is done by comparing the radii of each boss with the other two. Finally, after the boss with the smallest radius value is identified, its radius is changed to the desired value. Table 5.1 shows the algorithm to solve this design problem using design exemplars.

Table 5.1: Algorithm to find the boss with the shortest radius value in Figure 5.4

1. Find all the bosses in the model and tag them.
2. If a pair of bosses with tags are found, compare their radii and remove the tag on the boss with higher radius value and repeat step 2; Else if a pair of tagged bosses are not found goto 3.
3. If a boss with a tag is found, change the radius to 25 mm and exit the network; else exit the network.

While each operation that is required to be performed to achieve this task is represented using an exemplar, the nodes holding these exemplars are connected in a network to represent the algorithm, as shown in Figure 5.5.

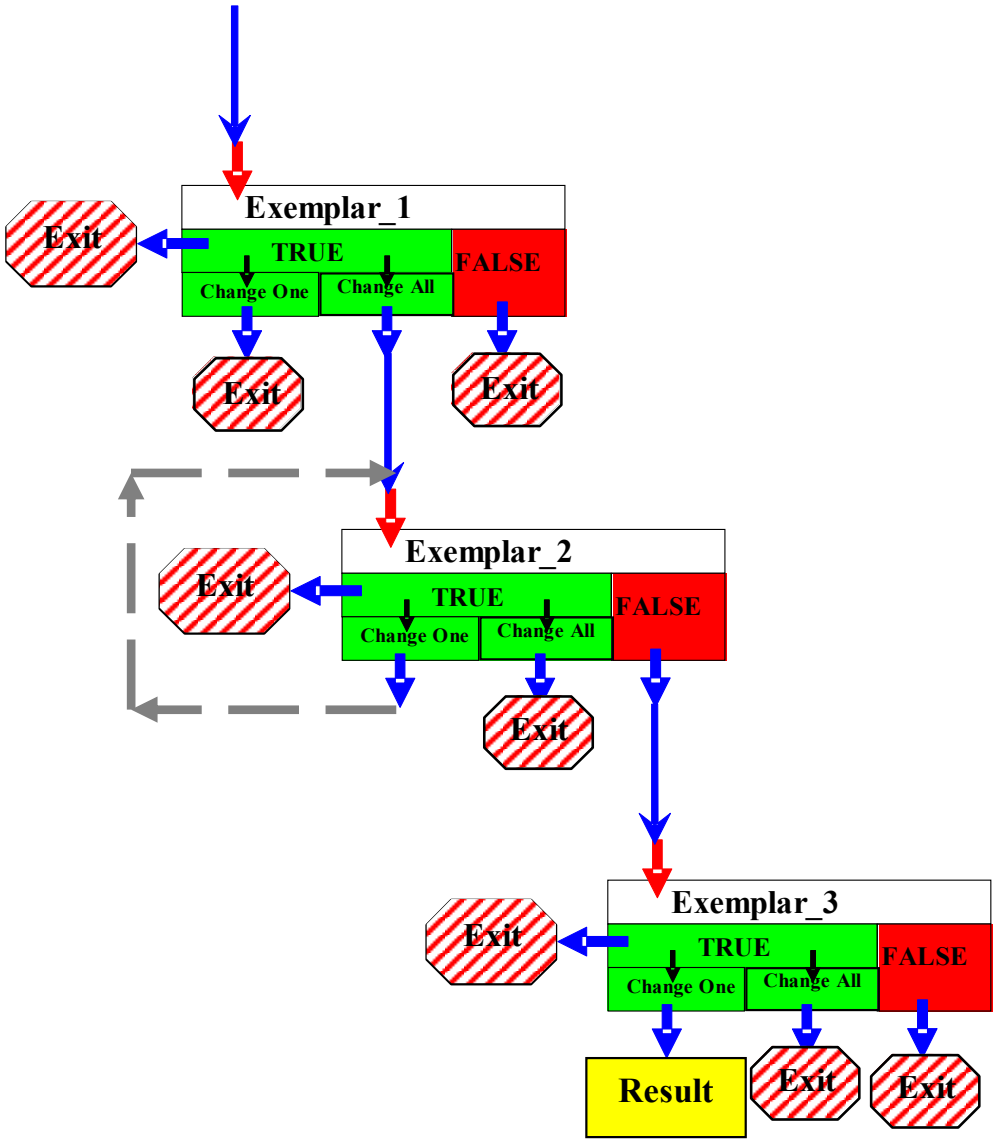


Figure 5.5: A Cyclic Network of exemplars representing the algorithm shown in Table 5.1

The model is first queried against the exemplar in the Exemplar_1 node. Since the model shown in Figure 5.4 has three untagged bosses in it; this check results in three matches --A, B, C. Now, the user has the option of choosing between tagging one boss at a time and tagging all at once. However, the algorithm suggests that the bosses be tagged all at once, therefore, the change all option at this node is selected. Hence, the matches found after the exemplar check at the first node are all tagged. The next exemplar check is performed at the Exemplar_2 node of the network, where a pair of tagged bosses is matched and the tag from the boss with greater radius is removed. It should be noted that a connection is

made from the change one port of the node to the node itself making this path cyclic. Therefore, when the model with the three tagged bosses is processed using this node, the bosses get untagged one at a time until a pair of untagged bosses is not found, in this case, after two iterations. The tagged boss that is found at the end of these iterations is the one with the smallest radius value. The next exemplar check is performed at the Exemplar_3 node of the network, and the radius of the boss that remained tagged at Exemplar_2 node is changed to desired value. A description of the actions performed in each cycle of the network nodes is shown in Table 5.2.

Table 5.2: Operations performed in each iteration of the nodes.

Iteration	Exemplar_1		Exemplar_2		Exemplar_3	
	Matches Found	Action	Matches Found	Action	Matches Found	Action
1	A, B, C	A, B, C tagged	AB, BA, AC, CA, BC, CB	• Identifies AB • Compares radius of A, B • Removes tag A	-	-
2	-	-	Two matches BC, CB found	• Identifies BC • Removes tag B	-	-
3	-	-	No match found	• Exemplar check returns false	One match, C, found	• Modifies the radius to the desired value

This process of identifying the boss with the smallest radius value and changing its radius to a desired value is shown in detain in the Figure 5.6.

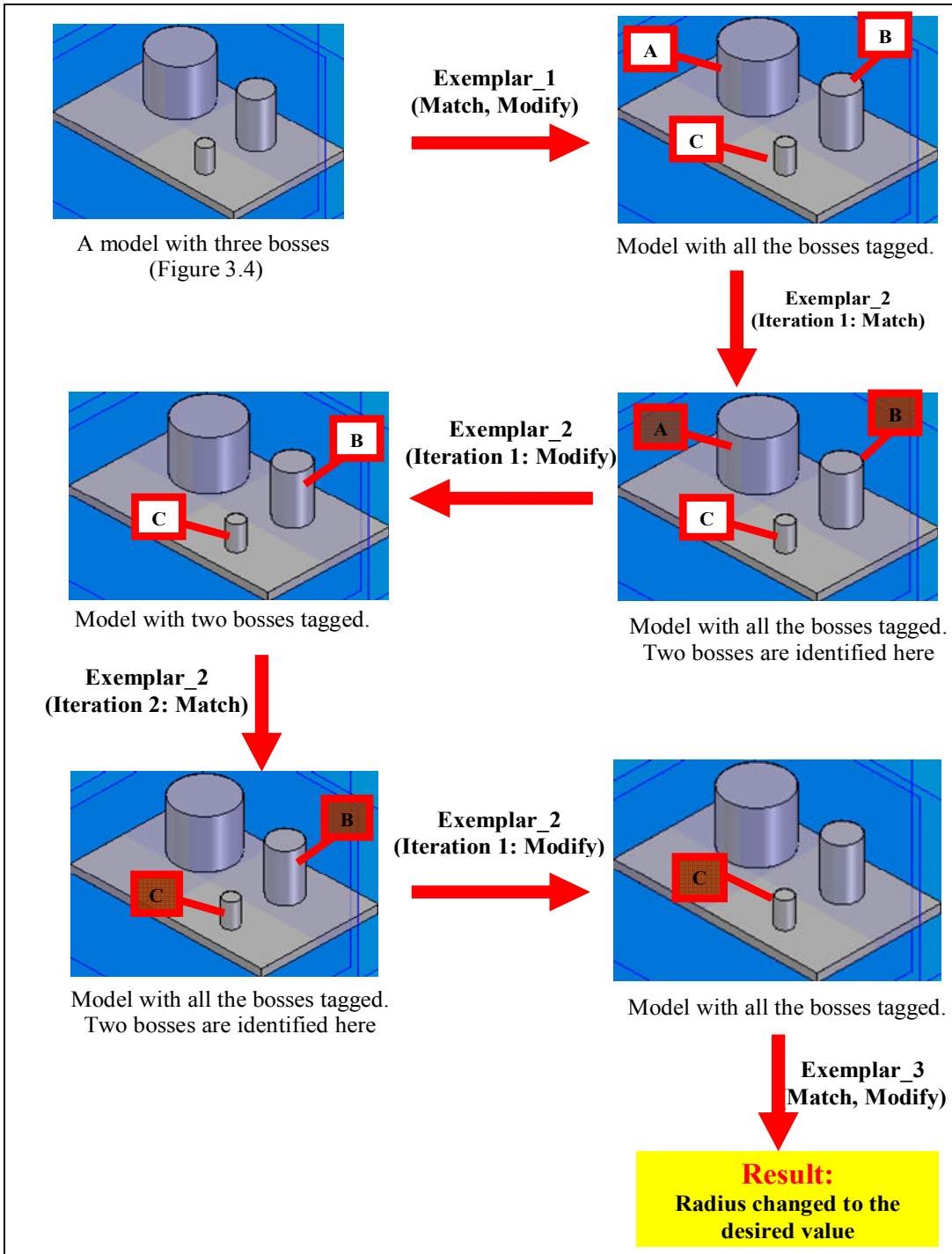


Figure 5.6: Changes the model undergoes while traversing the Dynamic Network

Dynamic Networks with Logical Connectives

Logical connectives may be used for representing complex design conditions in a design exemplar. The use of the logical connectives often results in a design exemplar that accommodates conflicting design conditions. The dynamic networking approach proposed here, also accommodates the use of exemplars composed by using logical connectives. For convenience, such exemplars will hereafter be referred to as the "hybrid exemplars" and the dynamic networks using these exemplars as the "Hybrid Networks". Figure 5.7 shows such a hybrid network formed by connecting four nodes.

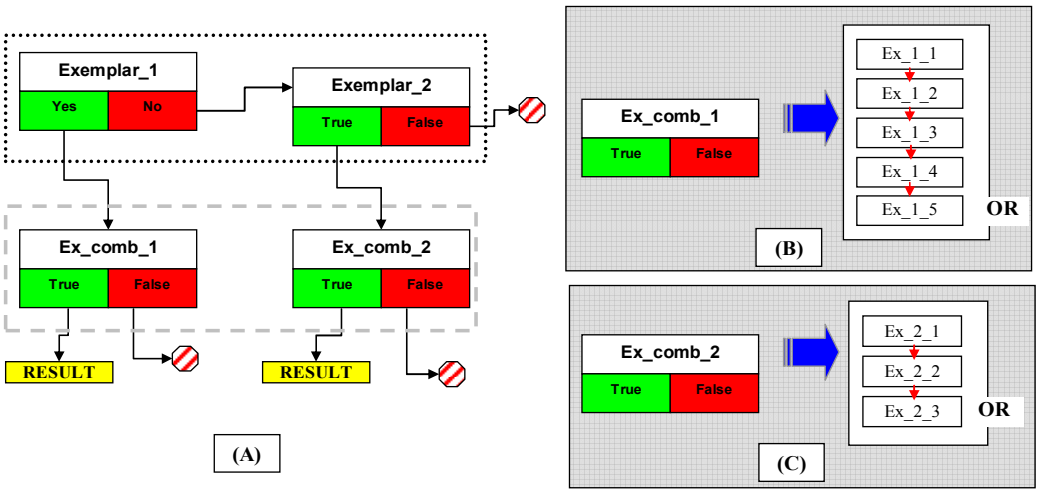


Figure 5.7: A Dynamic Network with exemplar formed using Logical Connectives

The nodes, Exemplar_1 and Exemplar_2, found in the first level of the network shown in Figure 5.7 are authored using entities and relations, while the exemplars Ex_comb_1 and Ex_comb_2, found in second level of this network are formed by combining other exemplars. As shown in Figure 5.7 (b) and Figure 5.7 (c), the exemplar in the node Ex_comb_1 is formed by combining five exemplars using the logical connective OR and the exemplar in node Ex_comb_2 is formed by combining three. However, it should be noted that, while Figure 5.7 incorporates hybrid exemplars only in the second level, these can be used at any level. As with any dynamic network, each node of the hybrid network is represented by a constraint problem, and a series of constraint problems represent the entire network.

The dynamic networking approach is presented as tool to the enable procedural processing of design data in the existing design exemplar system and adds a new dimension to its problem solving capabilities. Since, this approach is believed to facilitate operating geometric data in a fashion similar to looping and conditional branching in programming languages; it is also being presented as a step towards

the realization of a mechanical design visual programming language. However, the usefulness and performance of this approach in solving design problems needs to be evaluated. Case studies presented in the following sections of this chapter provide a validation for the hypothesis made in this research and also evaluate its usefulness in solving design problems.

Case Study: Gear Train Design

To establish the usefulness of the dynamic networking approach proposed in this chapter, a case study involving the design of a gear train is performed. The gear train problem introduced in Chapter 4 is considered for this purpose. An outline of the dividing method and the algorithm (Table 4.8) to achieve this design task is presented in chapter 4. Table 5.3 shows a modified algorithm for the dividing method that can be used for this gear train design problem in the design exemplar system.

Table 5.3: Algorithm for gear train design using exemplars

1. Introduce a pair of gears (Input Gear Pair) and a ratio parameter into the model; proceed to step 2.
2. Set the ratio of the gears to n:1 and identify the output gear; proceed to step 3.
3. Check the ratio of the last gear pair (will be called as Gear Pair- N); if it exceeds the maximum permissible gear ratio (6:1), proceed to step 4; else, proceed to step 7.
4. Introduce a new pair of gears into the model; proceed to step 5
5. Identify the last two gear pairs (Gear Pair-N and the newly added gear pair). Set the ratio of Gear Pair- N to 6:1and the ratio of the newly introduced gear pair to n: 1 (where n=n/6); proceed to step 5.
6. Identify the last two gear pairs and set the distance between the shafts for Gear Pair-N equal to the sum of the pitch circle radii of its two gears; Set the distance between the shafts for the newly introduced gear pair (in step 4) equal to the sum of the pitch circle radii of its two gears; proceed to step 3.
7. Remove the ratio parameter; exit.

Seven exemplars, each representing an operation performed in this algorithm can be used to solve this problem. A detailed description of operations performed by these exemplars: *AddGearPair, TagOutputGear, CheckRatio, AddNextLeveGears, DivideRatio, SetShaftDistance, RemoveRatio,* and their textual representation are presented in Table 5.4. For this problem, the desired input to output ratio is given as 60: 1 and the minimum radius of a gear is taken as 0.12cm.

Table 5.4: The Exemplars used to solve the Gear Train - Design Problem

Exemplar Description	Textual Representation Of The Exemplar
1. **AddGearPair:** Check to see if the model has a pair of gears; a. The exemplar will always be true (nothing to make it false) b. Then, a transformation action adds a new pair of gears and a *Ratio* parameter and set its value as 60.	**Alpha Match:** **Beta Match:** Parameter "Ratio" Circle "C1" Circle "C2" Line "C3" Line "C4" *Concentric* (C1, C3) *Concentric* (C2, C4) *Parallel* (C3, C4) *Tangent* (C1, C2) *Fixed* (Ratio)
2. **TagOutputGear:** Check to see if the model has a pair of gears and a ratio parameter; a. If yes, find the gear with the greater value of the pitch circle radius and tag it as the "*Output Gear*"; Set the radii of the pinion and the gear to 0.12 and Ratio * 0.12 respectively. b. If no match found, exit.	**Alpha and Beta Match:** Parameter "*Ratio*" Circle "C1" Circle "C2" *Tangent* (C1, C2) **Alpha Extract:** Parameter "r1" Parameter "r2" *Radius* (r1,C1) *Radius* (r2,C2) *Boolean expression* "CheckRadius"(r1<r2) **Beta Extract:** Parameter "r3" Parameter "r4" *Radius* (r3,C1) *Radius* (r4,C2) *Equation* "EqR3" (r3=0.12) *Equation* "EqR4" (r4=Ratio*0.12) **Beta Match:** *ID* "Output Gear" (C2)
3. **Check Ratio:** Identify the ratio parameter check if its value is less than 6.	**Alpha Beta Match:** Parameter "Ratio" *ID* (Ratio) **Alpha Extract:** *Boolean expression* "CheckRatio" (Ratio > 6)

Table 5.4: The Exemplars used to solve the Gear Train - Design Problem (Continued)

Exemplar Description	Textual Representation Of The Exemplar
4. AddNextLevelGears: Match the last gear pair of the series (identified using the "*Output Gear*" tag); a. If match found, add a new pair of gears (gear pair -2) such that, one gear of the this pair lies on the output shaft of the identified gear pair(gear pair - 1); Remove the "*Output Gear*" tag from the gear pair-1; Tag the second gear of gear pair -2 as the "*Output Gear*". b. If match not found, exit.	**Alpha and Beta Match:** Line "C2" Circle "C4" *Concentric* (C2, C4) **Alpha Match:** *ID* "output gear" (C4) **Beta Match:** Line "C5" Circle "C6" Circle "C7" *Concentric* (C2, C6) *Concentric* (C5, C7) *Parallel* (C2, C5) *Tangent* (C6, C7) *ID* "Output Gear" (C7)
5. DivideRatio: Find the last two gear pairs of the series (Identified based on the "*Output Gear*" tag); a. If match found, set the radius value of the gears in gear pair – 1 such that the ratio is 6 (to obey Rule 1).	**Alpha and Beta Match:** Parameter "Ratio" Line "C1" Line "C2" Line "C3" Circle "C4" Circle "C5" Circle "C6" Circle "C7" *Concentric* (C1, C4) *Concentric* (C2, C5) *Concentric* (C2, C6) *Concentric* (C3, C7) *Parallel* (C1, C2) *Parallel* (C2, C3) *Tangent* (C4, C5) *Tangent* (C6, C7) *ID* "Output Gear"(C7) **Alpha and Beta Extract:** Parameter "temp" **Alpha Extract:** *Equation* "eq_a" (temp=ratio); *ID* (C7) **Beta Extract:** *Equation* "eq_b" (ratio=temp/6) Parameter "r4b" Parameter "r5b" Parameter "r6b" Parameter "r7b" *Radius* "r4b" (C4) *Radius* "r5b" (C5) *Radius* "r6b" (C6) *Radius* "r7b" (C7) *Equation* "eq_r4b" (r4b=0.12) *Equation* "eq_r5b" (r5b=6*0.12) *Equation* "eq_r6b" (r6b=0.12) *Equation* "eq_r7b" (r7b=ratio*0.12)

Table 5.4: The Exemplars used to solve the Gear Train - Design Problem (Continued)

Exemplar Description	Textual Representation Of The Exemplar
6. SetShaftDistances: Find the last two gear pairs and the three shafts holding these gear pairs of the series (Identified based on the "*Output Gear*" tag); **a.** Set the distance between the first two shafts as D1= R1 + R2; Where D1 is the distance (parameter) between the shafts • R1 is the radius of the input gear in the first gear pair • R2 is the radius of the output gear in the first gear pair **b.** Set the distance between the first two shafts as D2= R3 + R4; Where D2 is the distance (parameter) between the shafts • R3 is the radius of the input gear in the first gear pair • R4 is the radius of the output gear in the first gear pair	**Alpha and Beta Match:** Parameter "Ratio" Line "C1" Line "C2" Line "C3" Circle "C4" Circle "C5" Circle "C6" Circle "C7" *Concentric* (C1, C4) *Concentric* (C2, C5) *Concentric* (C2, C6) *Concentric* (C3, C7) *Parallel* (C1, C2) *Parallel* (C2, C3) *Tangent* (C4, C5) *Tangent* (C6, C7) *ID* "Output Gear"(C7) **Beta Extract:** Parameter "D1" Parameter "D2" Parameter "R1" Parameter "R2" Parameter "R3" Parameter "R4" *Radius* "R1" (C4) *Radius* "R2" (C5) *Radius* "R3" (C6) *Radius* "R4" (C7) *Distance* "eq_D1" (D1=R1+R2) *Equation* "eq_D2" (D2=R3+R4)
7. Remove Ratio: Find the Ratio parameter **a.** Remove the Ratio parameter.	**Alpha Match:** Parameter "Ratio" *Fixed* (Ratio)

Seven dynamic exemplar nodes, each holding one of these exemplars listed in Table 5.4 are formed. These nodes are linked to form a dynamic network, such that the control flow from one node to the other is similar to the control flow form one step to the other in the dividing method algorithm. Such a dynamic network represent the algorithm discussed in Table 5.3 and is shown in Figure 5.8. For convenience, the nodes in this network are named after the exemplars they hold. Since the node holding

the *AddGearPair* exemplar represents the first step in the dividing method algorithm to solve this problem, it is the root node of this network, meaning that the process of gear design starts at this node.

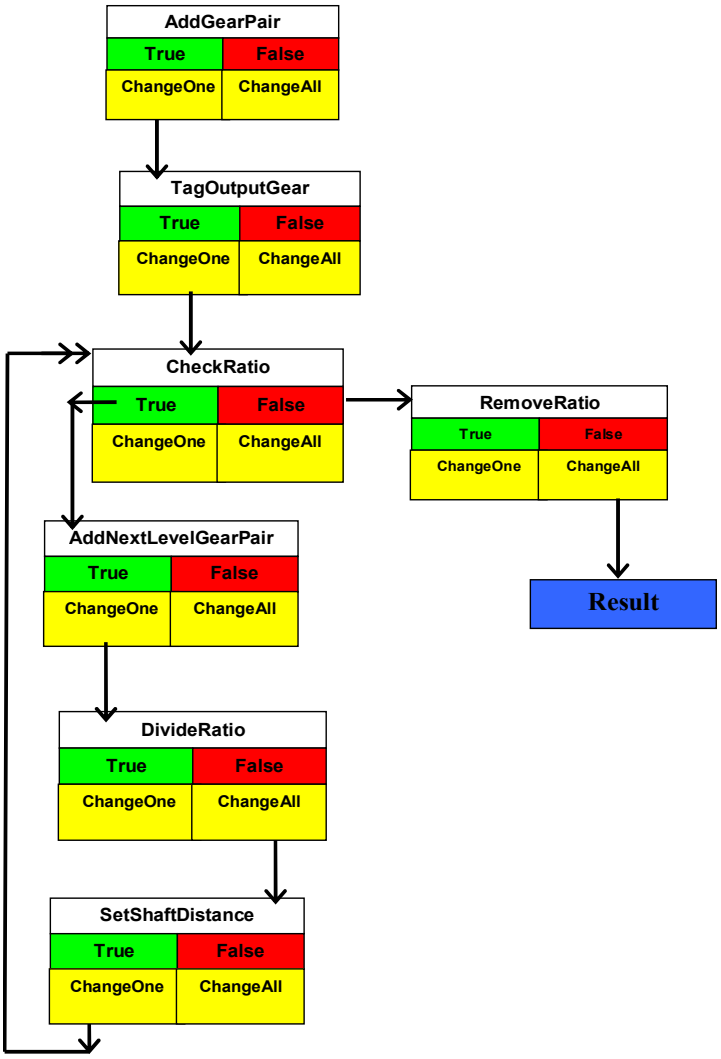


Figure 5.8: The Dynamic Network representing the Algorithm for the Gear Train Design using the Dividing Method:

As an initiator to the process, a pair of gears and a ratio parameter is introduced into the model at the first node (*AddGearPair*) of the dynamic network. The model, with a pair of gears and a ratio parameter is then traversed into the *TagOutputGear* node through the *ChangeOne* port of the *AddGearPair* node for processing. The radii of the two gears present in the mode are now compared and the gear with the larger pitch circle radius is tagged as the output gear. Also, the ratio parameter is set to the desired ratio. The model is then processed at the *CheckRatio* node, where the value of the *Ratio* parameter is checked. If

this value is found to exceed the maximum permissible gear ratio of *6:1*, the model is then processed at the *AddNextLevelGears* node. Here, a new pair of gears with the gear having larger radius, tagged as the output gear is introduced into the model. The model is then processed at the *DivideRatio* node, where the value of the ratio parameter is changed to (*ratio = ratio/6*). In order to maintain the desired speed ratio of *60:1* in the gear train, the ratio first gear pair is set to the maximum permissible gear set value (6:1) and the ratio of the output gear pair (the lastly introduced gear pair) to *ratio: 1* (here, the updated value of the ratio is considered). The model is now traversed into the *SetShaftDistance* node, where the shaft distances are such that meshing of gears is possible. A detailed description of the operations performed at each node of the dynamic network (Figure 5.8) and the evolution of the model as it traverses through the network is shown in Table 5.5.

Table 5.5: Figure showing the various operations performed at each node and the evolution of the gear train in Iteration 1

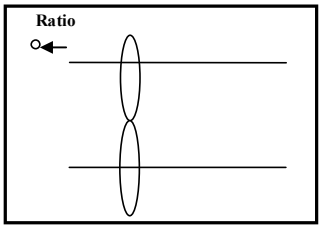
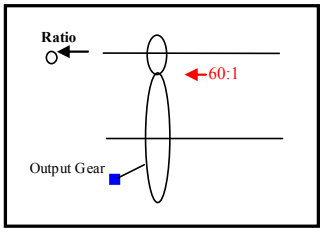
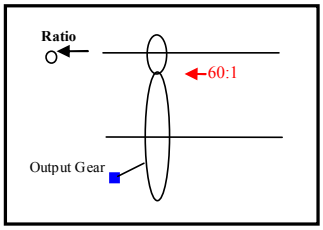
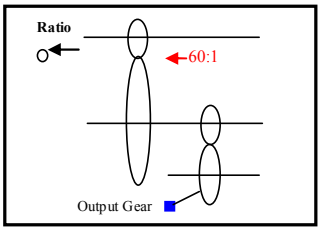
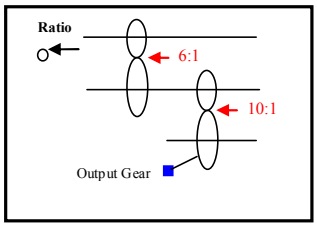
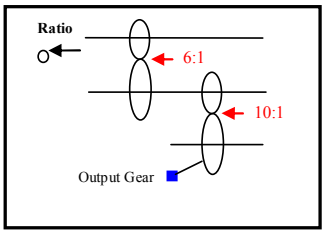
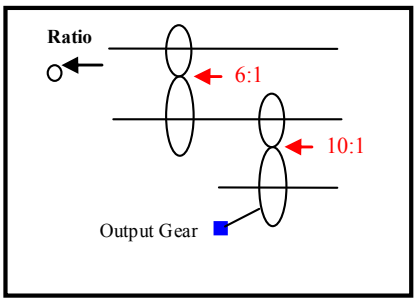
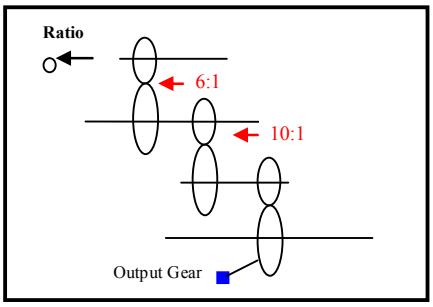
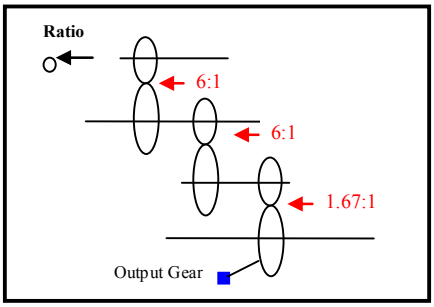
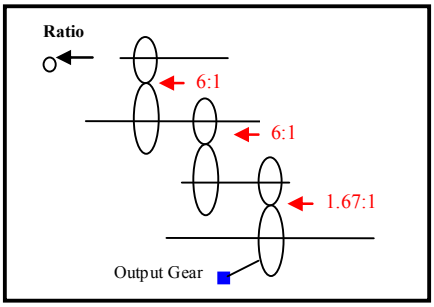
Dynamic Exemplar Node	Action performed at the Node	Gear Train Model
Add_Gears	• As an initiator to the process, the user draws a ratio with a fixed constraint. • A pair of gears with arbitrary gear ratio is added to the model.	
TagOutputGear	Find (Match) the gear pair and the ratio parameter introduced in the previous step. • Compare the radii of the two gears and tag the gear with the max. radius as the output gear • Set the value of the ratio parameter to (60:1)	
CheckRatio	Find (Match) the gear pair and the ratio parameter • Check if the ratio is less than the fixed value 10.	

Table 5.5 : Figure showing the various operations performed at each node and the evolution of the gear train in Iteration 1(Continued)

Dynamic Exemplar Node	Action performed at the Node	Gear Train Model
AddNextLevelGears	Find (Match) the gear pair and the ratio parameter • Add another gear pair (one tangent to the other) such that one gear lies on the output shaft of the previous gear pair. • Remove the "*Output gear*" tag from the second gear; Tag the last gear of the train as the "*Output Gear*"	
DivideRatios	Find (Match) the last two gear pairs based on the "*output gear*" tag, the shafts and the ratio parameter • Set the ratio of the first gear pair to 6:1 • Set the ratio of the second gear pair to (Ratio/6) :1 • Set the parameter Ratio = Ratio/6	
SetShaftDistance	Find (Match) the last two gear pairs based on the "*output gear*" tag, the shafts and the ratio parameter • Add a distance constraint on the first to shafts and set it to r1+r2 (r1, r2 are the radii of the gears) • Add a distance constraint on the second and the third shafts and set it to r3+r4 (r3, r4 are the radii)	

At the end of processing at the *SetShaftDistance* node, the model has two gear pairs such that the effective gear ratio is 60:1 and a ratio parameter whose value is 10. As indicated in the step 6 of the dividing method of the algorithm, this model is again processed at the *CheckRatio* node of the network. Since, the value of the ratio parameter exceeds the maximum permissible ratio value (*6:1*), the exemplar check at this node returns true and hence, the model is again processed at the *AddNextLevelGears*, *DivideRatios*, *SetShaftDistance* nodes. A detailed description of the operations performed at each node and the changes made to the model are presented in Table 5.6.

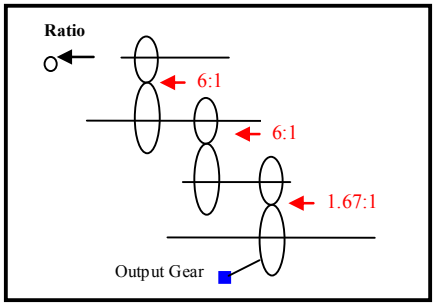
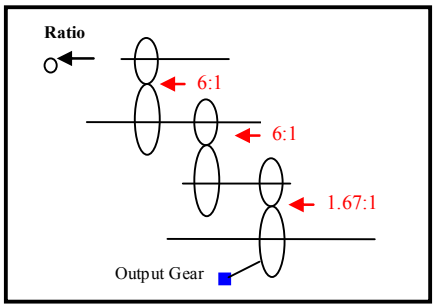
Table 5.6: Figure showing the various operations performed at each node and the evolution of the gear train in Iteration 2

Dynamic Exemplar Node	Action performed at the Node	Gear Train Model
CheckRatio	Find (Match) the gear pair and the ratio parameter • Check if the ratio is less than the fixed value 10.	
AddNextLevelGears	Find (Match) the gear pair and the ratio parameter • Add another gear pair (one tangent to the other) such that one gear lies on the output shaft of the previous gear pair. • Remove the "*Output gear*" tag from the second gear; Tag the last gear of the train as the "*Output Gear*"	
DivideRatios	Find (Match) the last two gear pairs based on the "*output gear*" tag, the shafts and the ratio parameter • Set the ratio of the first gear pair to 6:1 • Set the ratio of the second gear pair to (Ratio/6) :1 • Set the parameter Ratio = Ratio/6	
SetShaftDistance	Find (Match) the last two gear pairs based on the "*output gear*" tag, the shafts and the ratio parameter • Add a distance constraint on the first to shafts and set it to r1+r2 (r1, r2 are the radii of the gears) • Add a distance constraint on the second and the third shafts and set it to r3+r4 (r3, r4 are the radii)	

At the end of the second iteration, the model has a gear train of three gear pairs arranged such that the ratio between the input and output gears is set to 60. The value of the *Ratio* parameter is set to 1.67. When this model is again processed at the *CheckRatio* node as required by the step 6 of the algorithm in Table 5.3, since the value of the *Ratio* parameter is less then the maximum permissible gear set ratio (6: 1),

the check returns false. Hence, the model branches off into the node connected to the *false* port of the *CheckRatio* node. It is processed at the *RemoveRatio* node, where the ratio parameter is identified and removed from the model. Finally, a gear train of three gear pairs with the desired effective gear ratio is obtained. A description of the changes that occur in the third iteration can be seen in Table 5.7.

Table 5.7: Figure showing the various operations performed at each node and the evolution of the gear train in Iteration 3

Dynamic Exemplar Node	Action performed at the Node	Gear Train Model
CheckRatio	Find (Match) the gear pair and the ratio parameter • Check if the ratio is less than the fixed value 10.	
Remove Ratio	Find the ratio parameter • Remove the ratio parameter	

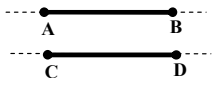
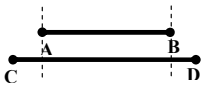
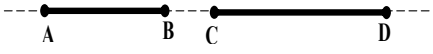
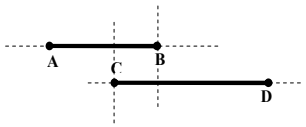
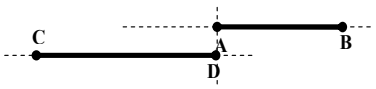
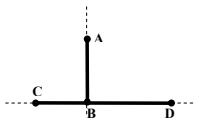
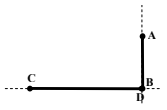
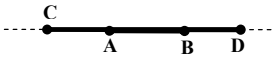
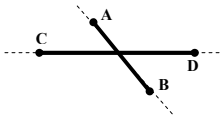
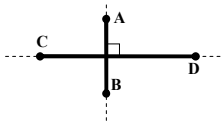
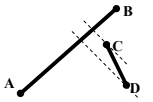
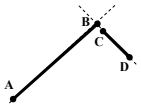
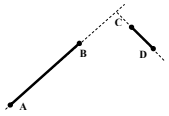
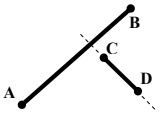
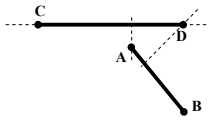
Observations

The dynamic network of exemplars has been successfully used to accomplish a design task that involves performing a series of operations on a model. The successful use of the dynamic network for designing a gear train has provides a validation for its use in design problems that require procedural processing. Hence, the dynamic network has provided a way for procedural processing of design data within the design exemplar system. While the looping functionality of the dynamic networks is demonstrated by the iterative use of *CheckRatio, AddNextLevelGears, DivideRatios, SetShaftDistance* nodes, the conditional branching ability is demonstrated at the *CheckRatio* node.

Case Study: Performance Evaluation

The dynamic networking of exemplars is presented to establish the hypothesis 2 of this research. In order to validate this hypothesis and to evaluate these networks in performing design tasks, a case study involving the determination of the shortest distance between two lines is presented here. For convenience, this study is limited to lines in a two-dimensional plane and of three mutually exclusive types -- parallel, intersecting and non-parallel lines. Within these three types, this case study will consider only the fifteen line configurations shown in Table 5.8.

Table 5.8: The line configurations considered for the Performance Evaluation Case Study

Type of Lines	Line Configurations	
Parallel Lines		
		
		
Intersecting Lines		
		
		
Non- Parallel Lines		
		
		

In order to calculate the distance between two lines in any of these configurations, an exemplar representing each case is composed. These exemplars will then be used to determine the distance between any two given lines (given the configuration of these lines is similar to one of the above fifteen). Given the design exemplars representing each of the above configurations and the configuration of the lines whose shortest is to be calculated is not known, such a problem can be dealt with in two ways; the first being a Logical Connective Approach where the fifteen exemplars representing all the listed line configurations are combined using an OR block and the second being the dynamic network approach proposed in this chapter. It should be noted that this study doesn't necessarily establish the accuracy of the solution obtained or the processing speed of the dynamic networks, as these factors depend to a large extent on external factors like the processor speed.

For the purpose of this case study, a problem involving the evaluation of the shortest distance between a pair of line segments in a design model having three line segments: ab, cd, ef, coincident with three lines: A, B, C as shown in Figure 5.9 is considered. As can be seen from this figure, the model has both parallel (ef, cd) and intersecting ((ab, cd) (ab, ef)) line segments. Also, the lines A, B, C are coincident with the line segments ab, cd, ef respectively. It should also be noted that the configurations of the three pairs of line segments (ab,cd), (cd, ef) and (ab, ef) can be found in the considered set of fifteen line configurations.

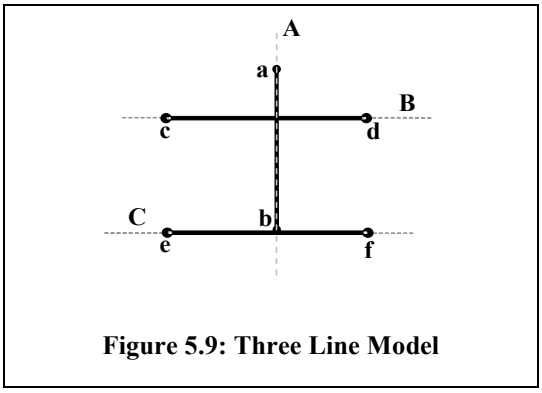


Figure 5.9: Three Line Model

This case study is preformed in two stages, while the first stage is performed to demonstrate the usability of the dynamic networking approach in solving design problems, the second stage is performed to validate the hypotheses made for the research questions 2.1 and 2.2 in Chapter 4. For the first stage of the case study, the shortest distance between a pair of line segments in the three line model of Figure 5.9 is evaluated using the pure network of dynamic nodes. As an alternative, a dynamic network using hybrid

exemplars is also used for this purpose. This example is extended further to perform the second stage of the case study, where the logical connective approach, which is traditionally being used for solving such problems, is considered. A comprehensive comparison is drawn between the three approaches (pure networks, dynamic networks with logical connectives and the logical connective approach), based on a complexity analysis and the number of solutions obtained in each case.

Stage 1: Dynamic Networking Approach

To solve the design problem in this case study a pure network formed by connecting the fifteen exemplars, introduced above may be used. Since, the task at hand does not require the iterative use of any of the exemplars; cyclic networks will not be considered. Three exemplars, for classifying a pair of line segments as parallel, intersecting or non parallel lines are additionally introduced into the dynamic network to reduce the number of constraint problems solved for obtaining a result. Nodes holding these exemplars form the first level of network, so that the given pair of lines is first classified into one of these types and subsequently queried using the exemplars of that type. The pure network used to find the shortest distance in the three-line model is shown in Figure 5.10.

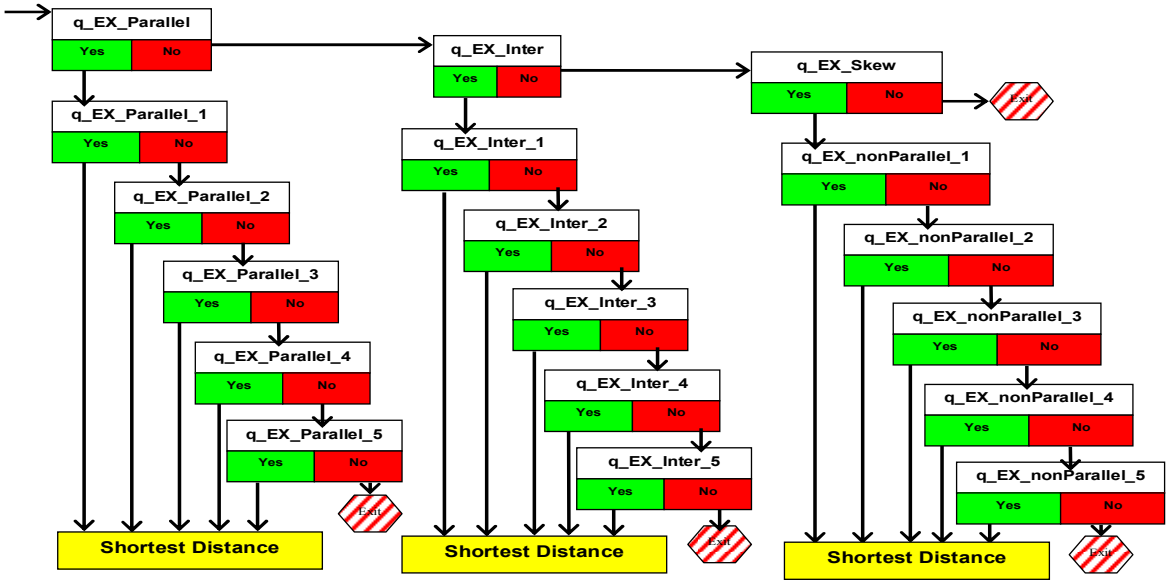


Figure 5.10: Pure Network for solving the two line problem

The alternative solution for solving this problem using the dynamic networking approach involves the use of exemplars that are formed by combining the exemplars representing the various configurations of

each type (parallel, non-parallel, intersecting) using logical connectives. Though this method involves the use of lesser number of nodes in the network, all the exemplars representing the fifteen line configurations mentioned are included. The hybrid network used for the purpose of this study is shown in Figure 5.11. The first level of this network is formed by exemplars used for classifying the lines into parallel, intersecting and non intersecting lines and the next level of the network is formed by nodes holding hybrid exemplars that evaluate the distance between them.

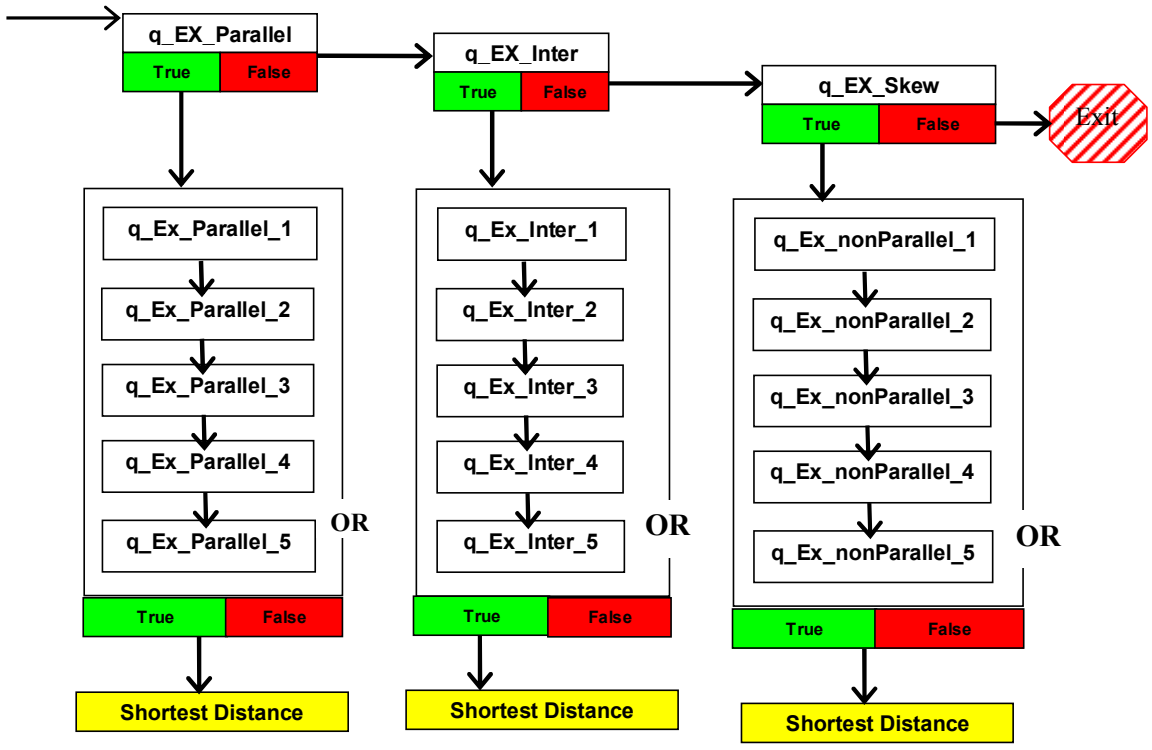


Figure 5.11: Hybrid Network to find the shortest distance between two lines

A keen observation at the dynamic networks used in this case study suggests that, when one of the nodes in the first level of the network returns a true, the control is branched into the sub-network connected to its true part, eliminating the possibility of querying the model against the other nodes in the same level of the network and thus eliminating the possibility of finding these configurations. Hence, it can be stated that the result obtained when the dynamic networking approach is used to solve the problem largely depend on the order of the nodes used. Since, the order of exemplars inside the OR blocks used in the second level of the hybrid network doesn't affect the number of constraint problems solved for obtaining the result; it is only the order of exemplars in the first level of the network that affects it. On the other side, the order of

the exemplars used in both the first and second level of a pure network effect the number of constraint problems solved for obtaining a solution. However, since there are fifteen exemplars in the second level of the pure network and considering all the possible sequences of these exemplars becomes a tedious task, this study is limited to the arrangement of exemplars only in the first level of the network. The three exemplars used in the first level of the network can be arranged in six different ways, which as mentioned earlier can affect the number of constraint problems. Each of these arrangements is discussed as a separate case in the following sections.

- CASE 1: (*q_Ex_parallel, q_Ex_intersect and q_Ex_skew)*: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_parallel, q_Ex_intersect and q_Ex_skew.
- CASE 2: *(q_Ex_intersect,* q_*Ex_parallel, q_Ex_skew)*: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_intersect, q_Ex_parallel, q_Ex_skew.
- CASE 3: *(Ex_skew, Ex_intersect and Ex_parallel)*: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_skew, q_ Ex_intersect and q_Ex_parallel.
- CASE 4: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_parallel, q_Ex_skew and q_Ex_intersect.
- CASE 5: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_intersect, q_Ex_skew and q_Ex_parallel.
- CASE 6: The sequence of nodes used in the first level of the networks for both the hybrid and pure networks is q_Ex_skew, q_Ex_parallel and q_Ex_intersect.

A detailed descripiton of the various checks peformed and the results obtained, when the model is queried against hybrid and pure networks, where the arrangement of the nodes in the first level of the network is as stated in each of these six cases is discussed below.

CASE 1: *(q_Ex_parallel, q_Ex_intersect and q_Ex_skew)*

In the pure network shown in Figure 5.10, nodes that are used to evaluate the distance between each of the three types: parallel, intersecting and non-parallel are grouped together to form sub-networks and are connected to the corresponding nodes in the first level network. The nodes in the first level of this network are linked such that the nodes holding the exemplars *q_Ex_parallel, q_Ex_intersect and q_Ex_skew* are connected in that order. When the three-line model shown in the Figure 5.9 is queried using

this network, it is first processed against the node holding the exemplar *q_Ex_parallel*. Since, the model has a pair of parallel line segments, (cd, ef) in it, this check returns true and the control branches off into the subsequent lower level network connected to true part of this node. The model is queried against each of these nodes sequentially until one of these checks returns a true and the shortest distance between the two parallel line-segments is found. However, it should be noted that the number of nodes processed before this value is obtained depends on the sequence of the nodes in the second level.

Similarly, when the model is queried against the hybrid network shown in Figure 5.11, it goes through a series of checks at the first level of the network until one of the exemplar checks return true, meaning that one of the three types of line configurations is found in the model. Now, the model is then branched into the second level of the network, where each node holds a hybrid exemplar formed by combining exemplars using OR blocks. It should be noted that the each exemplar in an OR block is processed as a separate constraint problem. Since, the hybrid network considered here has exemplars q_Ex_parallel, q_Ex_intersect, q_Ex_skew in that order representing parallel, intersecting, skewed / non-parallel lines respectively in the first level, the given model, when queried using this network is first checked for line segments that are parallel. As the line segments ab,cd in the three-line model are parallel, this check returns true at the *Ex_parallel* node, further the distance between these lines is evaluated using the exemplars in OR block connected to this node. The operations performed at the various nodes of the pure network shown in Figure 5.10, when the three-line model is processed using it are summarized in Table 5.9. The shortest distance between the line segments cd and ef is evaluated.

Though the order of pairing to calculate the shortest distance between two line segments does not affect the value obtained, both the pairs, for example (cd, ef), (ef, cd) in this case are considered different. This is because of the design exemplar representation takes the order of pairing into account while processing the model.

Table 5.9: Results obtained with the Dynamic Networks in Case 1

Pair of line segments	Pure Network Output		Hybrid Network Output	
	Line Configuration Identified	Distance	Line Configuration Identified	Distance
CASE 1: (Parallel + Intersecting + Skew)				
cd – ef	Parallel	D1	Parallel	D1
ef – cd	(Or) Parallel	D2	Parallel	D2
ab – cd	Check not performed	-	Check not performed	-
cd – ab	Check not performed	-	Check not performed	-
ab – ef	Check not performed	-	Check not performed	-
ef – ab	Check not performed	-	Check not performed	-

The results of querying the three-line model using both the dynamic networks discussed above are shown in Table 5.9. The order of pairing the line-segment is taken into consideration, and distance between each pair is evaluated separately. For example, (cd, ef) and (ef, cd) are identified as different pairs of lines and their distances evaluated as D1 and D2. Though the three-line model considered for this case study also has line-segment pairs that are not parallel to each other, it can also be seen from the that the dynamic network used here retrieves only the distance between the parallel lines while the checks for intersecting lines are not performed.

CASE 2: *(q_Ex_intersect,* q*_Ex_parallel, q_Ex_skew)*

While the algorithm for the dynamic network used remains unchanged, the sequence of exemplars used in the first level of the network for hybrid and pure networks is changed to *q_Ex_intersect,* q*_Ex_parallel, q_Ex_skew*. Hence, the results obtained when the three-line model of Figure 5.9 is processed using this network are different from the results obtained in Table 5.9. The model if first queried using the exemplars of the first level of the network, while the possibility of performing a check with the next node of the same level is decided dynamically. The results of processing the three line model using this network are shown in Table 5.10**.**

Table 5.10: Results obtained with the Dynamic Networks in Case 2

Pair of line segments	Pure Network Output		Hybrid Network Output	
	Line Configuration Identified	Distance	Line Configuration Identified	Distance
CASE 2: (Intersecting + Parallel + Skew)				
cd – ef	*Check not performed*	-	*Check not performed*	-
ef – cd	*Check not performed*	-	*Check not performed*	-
ab – cd	Intersecting	D1 = 0	Intersecting	D1 = 0
cd – ab	(OR)Intersecting	(Or) D2 = 0	Intersecting	D2 = 0
ab – ef	(OR)Intersecting	(Or)D3 = 0	Intersecting	D3 = 0
ef – ab	(OR)Intersecting	(Or) D4 = 0	Intersecting	D4 = 0

When the model is queried against the hybrid or pure network, it is first queried against the *q_Ex_intersect* exemplar in the first node of the network, and since the model has intersecting lines, this check returns true. The lines identified in this check are further queried against the next level of exemplars to find the shortest distance between them. These next level of checks is against a series of exemplars combined using an "OR" block in case of hybrid network and a second level dynamic network of exemplars in case of pure network. While the hybrid network returns the shortest distance between the line pairs (ab, cd), (cd, ab), (ab, ef), (ef, ab), the pure network returns the shortest distance between only one of these pairs as the control exits the network as soon as one of these pairs is identified.

CASE 3: *(Ex_skew, Ex_intersect and Ex_parallel)*

The sequence of exemplars used in the first level of the network for hybrid and pure networks is *Ex_skew, Ex_Intersect, Ex_parallel* in this case. Since the three-line model does not have any skewed lines in it, the first check preformed at the node with *q_Ex_skew* returns false and the control moves further in the same level. The next exemplar check on the model is preformed at the node holding the exemplar *q_Ex_intersect*. The line-segment pairs (ab, cd), (cd, ab), (ab, ef), (ef, ab) are identified and the model is further processed in the sub-network connected to the true part of this node. While the pure network returns the distance between one of these pairs, the hybrid network returns the distance between all four of them.

Table 5.11: The checks performed and the results obtained in Case 3.

Pair of line segments	Pure Network Output		Hybrid Network Output	
	Line Configuration Identified	Distance	Line Configuration Identified	Distance
CASE 3: (Skew + Intersecting + Parallel)				
cd – ef	Check not performed	-	Check not performed	-
ef – cd	Check not performed	-	Check not performed	-
ab – cd	Intersecting	D1 = 0	Intersecting	D1 = 0
cd – ab	(OR) Intersecting	D2 = 0	Intersecting	D2 = 0
ab – ef	(OR) Intersecting	D3 = 0	Intersecting	D3 = 0
ef – ab	(OR) Intersecting	D4 = 0	Intersecting	D4 = 0

It can be seen from Table 5.11 that the results obtained after processing the model using this network are identical to results obtained in Case 2. A similar analysis is preformed using the networks, where the node sequence in the first level is as identified in Case 4, Case 5 and Case 6. The solutions obtained in each of these cases are shown in Table 5.12.

Table 5.12: The pairs of line-segments identified in Case 4, Case 5, Case 6.

Pair of line segments	Pure Network Output		Hybrid Network Output	
	Line Configuration Identified	Distance	Line Configuration Identified	Distance
CASE 4: (Parallel + Skew + Intersecting)				
cd – ef	Parallel	D1	Parallel	D1
ef – cd	(Or) Parallel	(Or) D2	Parallel	D2
ab – cd	Check not performed	-	Check not performed	-
cd – ab	Check not performed	-	Check not performed	-
ab – ef	Check not performed	-	Check not performed	-
ef – ab	Check not performed	-	Check not performed	-

Table 5.12 : The pairs of line-segments identified in Case 4, Case 5, Case 6 (Continued)

Pair of line segments	Pure Network Output		Hybrid Network Output	
	Line Configuration Identified	Distance	Line Configuration Identified	Distance
CASE 5: (Intersecting + Skew + Parallel)				
cd – ef	Check not performed	-	Check not performed	-
ef – cd	Check not performed	-	Check not performed	-
ab – cd	Intersecting	D1 = 0	Intersecting	D1 = 0
cd – ab	(OR)Intersecting	(Or) D2 = 0	Intersecting	D2 = 0
ab – ef	(OR)Intersecting	(Or)D3 = 0	Intersecting	D3 = 0
ef – ab	(OR)Intersecting	(Or) D4 = 0	Intersecting	D4 = 0
CASE 6: (Skew + Parallel + Intersecting)				
cd – ef	Parallel	D1	Parallel	D1
ef – cd	(Or) Parallel	(Or) D2	Parallel	D2
ab – cd	Check not performed	-	Check not performed	-
cd – ab	Check not performed	-	Check not performed	-
ab – ef	Check not performed	-	Check not performed	-
ef – ab	Check not performed	-	Check not performed	-

The results of this case study indicate that dynamic networks may be used to solve design problems. However, it is also clear that the set of results obtained in the case of a pure network is smaller than the hybrid networks. This aspect of the pure networks however is useful in obtaining a refined solution set. Hence, it can be said that the dynamic networks may be used in conjunction with the logical connectives to obtain a broader solution set. While, the stage 1 of this case study has established the usability of the dynamic networking approach for solving design problems, it is extended further to evaluate the performance of these networks. The performance of these networks is gauged based on a complexity analysis in stage 2 of this case study.

Stage 2: Performance Evaluation of the Dynamic Networking Approach

It is often possible that a computational problem is solved using different algorithms. Selecting the algorithm that solves the problem in the shortest time or by using the least number of system resources is of great importance for good programming. The complexity of an algorithm is considered as an important measure in choosing the algorithm to be used to solve the problem. It is defined as a measure of the tasks performed for achieving a function [McCabe. et. al., 1989]. In other words, it is a value Figure showing the dependence of the amount of data needed for an algorithm and the time required or number of arithmetic/logical operations [Summers. et. al., 2003]. The complexities of the various algorithms, that may be used to solve a design problem are considered as a measure to rank them. Problems similar to the three-line model in this case study can be solved using the logical connectives approach. Hence, the big "O" complexity of the two dynamic network algorithms discussed in stage 1 are compared with the logical connectives algorithm, which is traditionally used for solving similar design problems. For this purpose, the complexities of these algorithms in the worst and best case scenarios are estimated and compared. While, the worst case scenario is identified as a situation where the algorithm has to perform the maximum number of exemplar checks on the model, the situation where the minimum number of exemplar checks is done is the best case scenario. In other words, the longest path followed in an algorithm to produce a result corresponds to the worst case scenario, while the shortest path followed to produce a result corresponds to the best case scenario.

Complexity: The Logical Connectives Method

In the logical connectives approach used to solve this problem, the exemplars to find the shortest distance between each of the fifteen line-segment pairs shown in Table 5.8 are grouped together using an OR block. The three-line model (shown in Figure 5.9) considered for this study is queried against this OR block. A set of constraint problems, each representing a design exemplar in the OR block is evaluated before a final result is obtained. Each constraint problem/exemplar retrieves the corresponding match found in the model. Hence, all line configurations represented by the exemplars included into the "OR" block are identified and their distances evaluated.

In order to perform a complexity analysis, the possible shortest and longest paths that the model may trace before a final result is obtained are identified. These paths represent the best and worst case scenarios based on the number of constraint problems solved before a final result is obtained. Figure 5.12

shows the longest and shortest paths that may be traced by the model within the OR block. The path taken in the worst case scenario, where the maximum number of constraint problems are solved before obtaining a result is shown using a solid red line and the best case scenario, where the minimum number of constraint problems is solved is shown using a solid blue line.

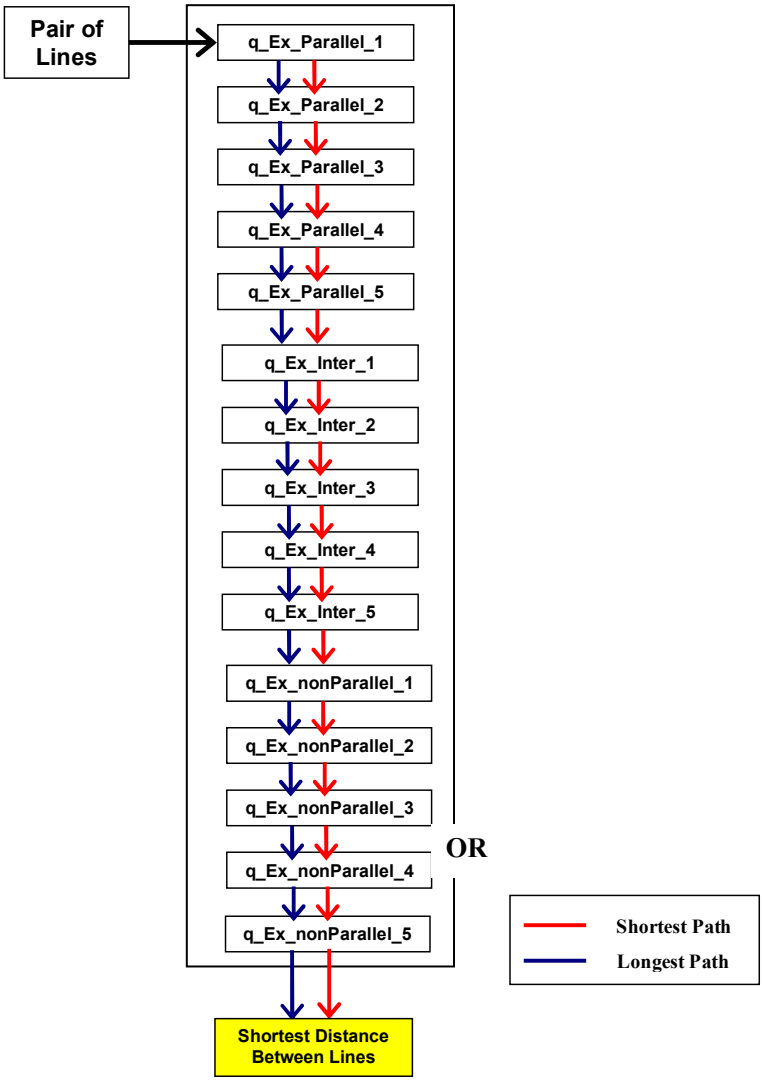


Figure 5.12: The shortest and longest paths followed in algorithm

Figure 5.12 shows that the possible longest and shortest paths representing the worst and best case scenarios that may arise are the same. This is because of the inherent nature of the OR blocks, where, the model is queried against all the exemplars included into it. Since, all these exemplars (constraint problems) are checked irrespective of the outcome of the previous checks, the running time of this algorithm is

expected to be linear. The numbers of constraint problems solved in this algorithm are of the order of *N*, where, N is the total number of exemplars included into the OR block. Hence, the Big (O) complexity of this algorithm is *O(N)*.

Complexity: Dynamic Networks with Logical Connectives (Hybrid Networks)

Logical connectives can be used to combine exemplars representing simple design problems to form an exemplar that can be used to represent a more complex problem. When such an exemplar is used to query a design mode, each of the exemplar within the OR block is evaluated as a separate constraint problem. Hence, in the Hybrid network discussed in stage 1 of this case study, each node in the second level of the network represents five constraint problems corresponding to the five design exemplars in the OR block. When a model is queried against such a node, it is queried against exemplar of that node. Figure 5.13 shows the dynamic network (hybrid) used in this case study with the shortest and the longest paths that may be taken by the model before a final result is obtained indicated using solid red and blue lines respectively. It can be seen from this figure that, though the number of constraint problems solved in worst and best cases are different, the order of complexities is still linear.

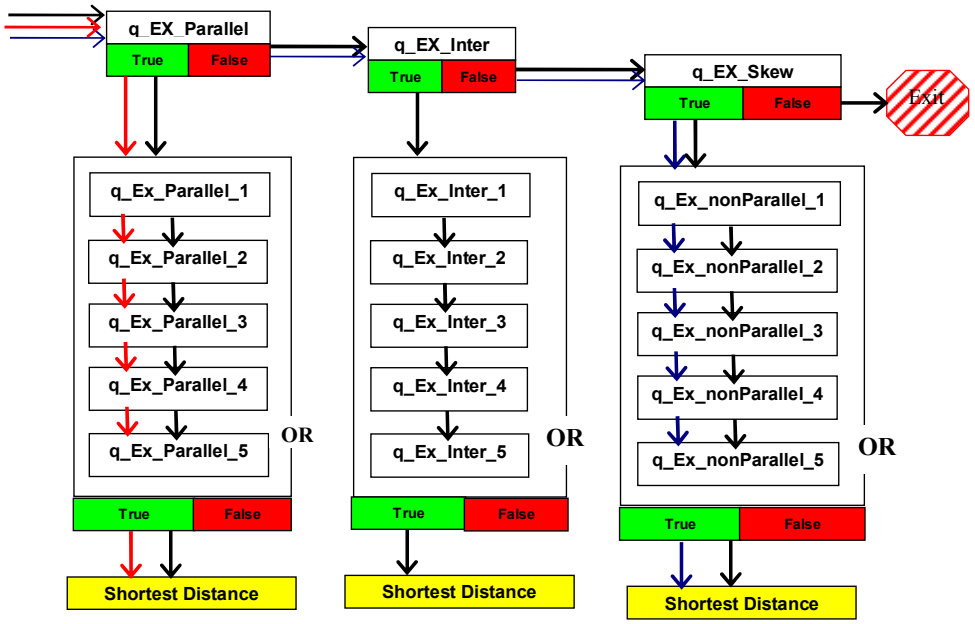


Figure 5.13: Shortest and Longest Paths in a Hybrid Network

Comparing the number of constraint problems solved here with the logical connectives approach clearly shows that the number of exemplar checks done before a final result is quite less in case of hybrid network. The running time of this algorithm may also be considered to be linear as the branching takes place only in the first level of the network. The total number of checks performed before a result is obtained is of the order of N/3. Hence, the complexity of this algorithm as per the big "O" notation may be shown as O(N/3) (since, N/3 << N).

Complexity: Pure network

The worst and best case scenarios where the highest and the least number of constraint problems are solved before a result is obtained are shown using solid red and blue lines in the Figure 5.14. To find the complexity of the pure network, it can be viewed as a binary tree of dynamic nodes. Hence, it can be said that the complexity of this network is of the order of ln (N).

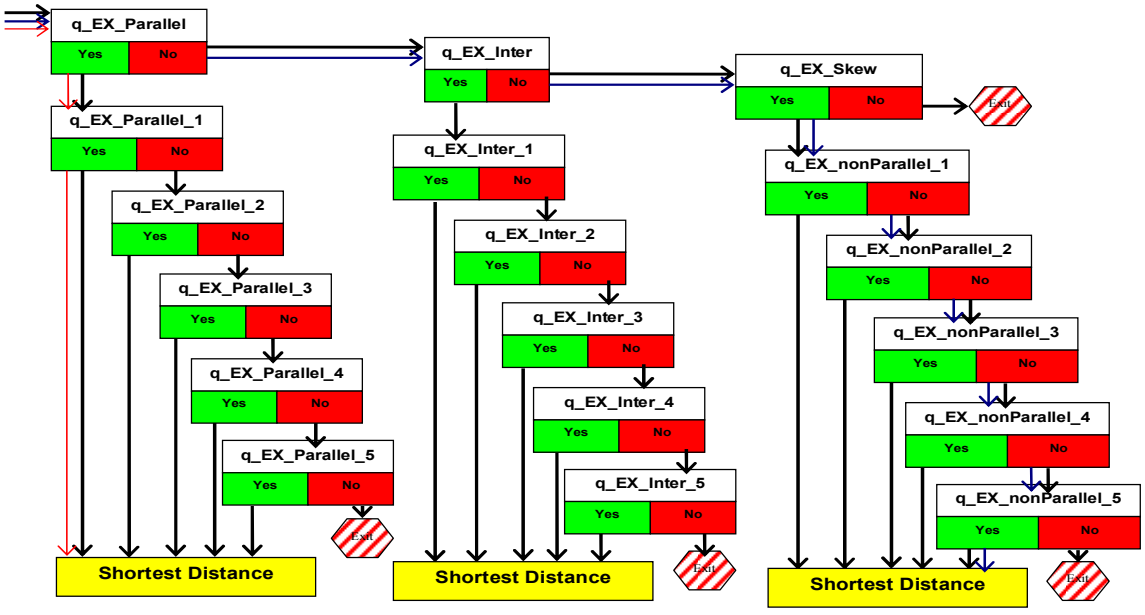


Figure 5.14: The Shortest and Longest Path in a Pure Network

Based on the number of number of constraint problems solved in the worst and best case scenarios for each algorithm presented above, the order of big "O" complexity is calculated. The complexities of each of the algorithms discussed in this case study are presented in Table 5.13. The complexity of the Logical Connectives approach is O (N), whereas the complexity of hybrid network is found to be O(cN) where, constant c = 1/3. A reduction in the complexity can be noticed from Logical Connectives approach

to hybrid network. This means that the computational time required for processing the problem using hybrid network is lesser than the time for processing the problem by the logical connectives approach. Similarly, the processing time required for solving the problem using pure network, is further low when compared to hybrid network.

Table 5.13: Big (O) Complexity for each algorithm

	Logical Connectives Approach		Dynamic network (Hybrid Network)		Dynamic Network (Pure Networking)	
	No. of Constraint Problems	Complexity	No. of Constraint Problems	Complexity	No. of Constraint Problems	Complexity
Complexity Order		N		N		ln (N)
Worst Case Scenario	15	O (N)	8	O(cN)	8	O(cN)
Best Case Scenario	15	O (N)	6	O (cN)	2	O (1)

From this case study, it is clear that though the three approaches used in this case study, do not yield the same results; they yield valid results when a model is queried against them. It is observed form the results presented in Table 5.13 that the complexity of the pure networks approach is far less when compared to the dynamic networks and further less than the logical connectives approach. When compared with the logical connectives, the hybrid network approach is observed to reduce the number of constraint problems solved to some extent by not performing checks in the same level, as soon as one check returns true. The pure network approach has further reduced this before the result is produced. Hence, it is observed that the number of general exemplar checks required to produce a result decreases with increased branching. It should be noted that the performance is not evaluated based on a time study or bench marked against the results of the existing networking approach. Instead, this analysis is presented as an evaluation of the effectiveness of the approach.

CHAPTER 6

CONCLUSIONS

The main objective of this research is to develop a visual programming language using the design exemplar representation, which then can be used by mechanical engineers to develop design automation applications without the need to learn high-level textual programming languages. Based on the level of programming expertise and mechanical engineering domain knowledge, this language can be used by three classes of users: Programmers, Application Developers, and End Users [Nackman. et. al., 1986]. Programmers can operate at the core level of this language in a C++ environment to develop or include new primitives, solvers (reasoners) and programming constructs. However, this requires considerable amount of expertise and knowledge in C++. As the second type of user, application developers may create applications that can be used for geometric data processing to automate a design process. These applications can vary from a large application that spans an entire domain to a smaller application that eases a specific task of the domain. For example, the development of a feature recognition system that facilitates the identification of the various components of a automobile engine, though the development of such a system is not yet established, is believed to be possible. Application developers typically do less amount of programming (textual) at the core level, and much of the programming is done in the visual programming environment presented in this research. Finally, end users typically are not familiar with programming but use software applications developed by the application developers to accomplish certain tasks. It should be noted that the application developers themselves can sometimes be the end users, illustrating the power of visual programming languages to bridge the gap between user and developer.

The design exemplar VPL proposed in this thesis makes use of the declarative design exemplar representation found in the literature [Bettig. 1999] [Summers. 2004]. While, the programming constructs and the rules that govern the visual sentence authoring remain the same as the rules for exemplar authoring, the design exemplar system is further enhanced to support a procedural programming construct named "Dynamic Network". This approach adds a new dimension to the design exemplar system, by allowing the automation of the problem solving process.

Q 1: Can the Design Exemplar System be developed into a Visual Programming Language?

- The design exemplar system provides visual representation of design using unique visual representations of the entities and constraints present in a model, while allowing to express the model by visual interaction. Hence, it is believed that the design exemplar system can be extended into a visual programming language.

Q 1.1: What components of a visual programming language are found in the existing Design Exemplar Technology?

- A clearly defined set of icons and iconic system is present in the existing design exemplar system.

Q 1.2: What components of a visual programming language are missing from the Design Exemplar Technology?

- The current design exemplar system does not support constructs for looping and conditional processing of data.
- The exemplar system lacks a Compiler.

Q 2: Can a procedural approach be employed in a Visual Programming Language for Mechanical Design?

- Yes, the procedural approach can be used in a VPL for Mechanical Design.

Q 2.1: Can this procedural approach be developed to extend the exemplar to support conditional branching?

- The design exemplar can be extended to support conditional branching.

Q 2.2: Can a procedural approach be developed to extend the exemplar to support looping?

- The proposed procedural approach in the design exemplar system can be extended to support conditional branching.

Q 2.3: How does the new approach affect the complexity of solving?

- When compared to the traditional approach, the complexity of solving is found to be reduced.

Q 2.4: How does the new approach affect the number of solutions obtained?

- The number of solutions obtained is reduced.

Contributions

The design exemplar has been presented as a data structure useful for representing design data which has been used to form a CAD query language [Divekar. et. al., 2003], to create a feature recognition system [Venkataraman. 2000], and to capture design validation rules for manufacturing analysis [Summers. 2004]. However, the true potential of the design exemplar system, as envisioned, has not yet been realized. The possibility of using the design exemplar in developing applications useful for mechanical engineers is analogous to the use of programming languages for developing software applications. Since, the design exemplar system makes use of objects like entities and constraints to represent design data, this research has tried to explore the possibility of developing the design exemplar system into a visual programming language. As an initial step towards this ultimate goal, the design exemplar system has been compared with a visual programming language to identify the similarities and evaluate the possibility of developing the design exemplar system in to a visual programming language. This has helped to identify the components of visual programming language that seem to be missing form the design exemplar system, specifically, a compiler and programming constructs for looping and conditional branching.

Though the declarative nature of the design exemplar representation was found adequate for querying and pattern matching applications, it was found to be inadequate for developing design automation applications where operations must be performed in a specific sequence. To address this inadequacy, the design exemplar system was enhanced to support procedural processing of design data using the "Dynamic Networking" of exemplars. Since, the primary objective of this research is to develop the design exemplar system into a programming language, not merely to query design information but to develop applications that can be used for geometric processing, it was identified that such a programming language be procedural in nature. The dynamic networking approach also provides for looping and conditional branching operations in a visual language perspective, enabling it to handle large sets of operations. The ability of looping is found to be very important in situations that require performing a set of operations in an iterative fashion until a terminal condition is met. or when the use of an exemplar is subjected to a condition.

Future Work

This research has helped to identify the various aspects of a programming language that need to be included into the existing design exemplar system to develop it into a visual programming language. Though a few of these aspects have been addressed in this research, the remaining is determined to be out of scope and are left for future work. Following are some aspects of the existing design exemplar system that need to be improved or addressed in order to realize the Design Exemplar Visual Programming language:

1. Develop methods to avoid infinite looping in dynamic networks.

   The dynamic networking approach proposed in this thesis can sometimes cause the process to get into an infinite loop. Getting into an infinite loop can cause unrecoverable loss of information due to system crash. To avoid such loss in information and to use the dynamic networks to their full potential, ways to address this issue should be devised and implemented.

2. Limited by the solver.

   The solving ability of the solvers currently supporting the design exemplar system is limited. For example, the current solvers do not support inequality constraints and cannot problems involving the evaluation of conditions with *less than* or *greater than* condition. In order to develop the design exemplar system into a full-fledged visual programming language for mechanical design, these issues related to the solver should be addressed.

3. Development of the Compiler

   The development of a complier, as stated already, is essential for the development of the design exemplar system into a visual programming language programming language. The development of a compiler essentially requires the development of an icon interpreter, a parser and a code optimizer.

LIST OF REFERENCES

Abacus., “Web: http://www.hks.com/”, ABAQUS, Inc., 2006.

Article on Mould Making using a 3D modeling software., “Web: http://www.moldmakingtechnology.com/articles/”, MoldMaking Technology Online, 2006.

A. Borning., “The programming language aspects of THINGLAB, A Constraint Oriented – Simulation Laboratory”, ACM Transactions on Programming Languages and Systems, 3(4), p252- 387, 1981.

A. C. Starling., “Performance-Based Computational Synthesis Of Parametric Mechanical Systems”, St John’s College, Cambridge University Engineering Dept,2004.

A. C. Ward, W. P. Seering., "Quantitative Inference in a Mechanical Design Compiler." Journal of Mechanical Design 115(1), pp: 29-35. 1993.

A. D. N. Edwards., “Visual programming languages: the next generation?”, ACM SIGPLAN Notices archive, Vol. 23, Issue 4, 1988.

A. Divekar, J. D. Summers., "The Design Exemplar: A Foundation for a CAD Query Language", Design Engineering Technical Conferences, ASME, DETC-2003, Chicago, IL, CIE-48228, 2003.

A. Divekar, J. D. Summers., "Logical Connectives for a CAD Query Language: Algorithms and Verification", Design Engineering Technical Conferences, ASME, DETC-2004, Salt Lake City, UT, CIE-57787, 2004.

A. Fish, J. Flower, J. Howse., “The semantics of augmented constraint diagrams”, Journal of Visual Languages and computing, p541-573, March 2005.

Ansys.,“Web: http://www.ansys.com/default.asp“, Ansys, Inc., 2006.

A. Paoluzzi, V. Pascucci, M. Vicentino., “Geometric Programming: A Programming Approach to Geometric Design”., ACM Transactions on Graphics, Vol. 14, No. 3, p 266-306, 1995.

A. V. Aho, R. Sethi, J. D. Ullman., “Compilers: Principles, Techniques, and Tools”, Bell Laboratories, 1986.

B. A. Myers., “Visual Programming, Programming by Example, and Program Visualization: A Taxonomy”, Human Factors in Computing Systems, p: 59 -66, 1986.

B. Bettig,V. Bapat,B. Bharadway., "Limitations of Parametric Operators for Supporting Systematic Design", Design Engineering Technical Conferences, ASME, DETC-2005, Long Beach, CA, DTM-85165, 2005..

B. Boshernitsan, M. S. Downes., “Visual Programming Languages- A Survey”, Report No. UCB/CSD -04-1368, Computer Science Division, EECS, University of Berkeley, Berkely, CA, 2004.

B. Hoffmann, M. Minas., "Towards Generic Rule-Based Visual Programming," p: 65, IEEE International Symposium on Visual Languages (VL'00), 2000.

C. A. M. Grant., “Visual Language Editing Using a Grammar-Based Visual Structure Editor”, Journal of Visual Languages and Computing, Vol. 9, p: 351- 374, 1998.

C. Frasson., M. Er-radi:, "Principles of an Icons-Based Command Language", in Proc. of ACM SIGMOD '86, Int'l Conf. on Management of Data, pp. 147-151, Washington, 1986.

CosmosWorks., “Web: http://www.solidworks.com/pages/products/cosmos/cosmosworks.html”, SolidWorks Corporation, 2006.

DFMA., “Web: http://www.dfma.com/publications/index.html”, Boothroyd Dewhurst, Inc., 2006.

D. Koelma, R. V. Balen, A. Smeulders., “SCIL-VP: a multi-purpose visual programming environment”, Symposium on Applied Computing, p 1188 – 1198, 1992.

E. K. Antonsson., “The Potential for Mechanical Design Compilation”, Research in Engineering Design, p: 191-194, 1997.

E. Post., “Formal reductions of the general combinatorial decision problems”, American Journal of Mathematics, 65, pp: 197-268, 1943.

F. Lakin., “Visual Grammars For Visual Languages”, AAAI87, Sixth National Conference on Artificial Intelligence, pp: 683-688, 1987.

G. Costagliola, V. Deufemia, F. Ferrucci, C. Gravino., “Using Extended Positional Grammars to Develop Visual Modeling Languages”, ACM International Conference Proceeding Series; Vol. 27, p 201 - 208, 2002.

G. Costagliola, A. Delucia, S. Orefice, G. Polese., “A Classification Framework to Support the Design of Visual Languages”, Journal of Visual Languages & Computing, Vol. 13, p 573-600, 2002.

G. D. Penna, B. Intrigila, S. Orefice., “An environment for the design and implementation of visual applications”, Journal of Visual Lanuguage and Computing, 15(6) pp: 439-461, 2004.

G. Nelson., “JUNO, A Computer Based Graphics System”, Computer Graphics, 19(3), p235-243, 1985.

G. Reddy, J. Cagan, "An Improved Shape Annealing Algorithm For Truss Topology Generation," ASME Journal of Mechanical Design, Vol 117, No. 2(A). pp. 315-321, 1995.

G. Steele, G. J. Sussman., “CONSTRAINTS – A Language For Describing Almost Hierarchical Descriptions”, Artificial Intelligence, 14(1) ,p 1-39, 1980.

G., Stiny, W. J. Mitchell., “The Palladian grammar. Environment and planning”, Planning & Design 5, pp: 5–18, 1978.

H. Koning, J. Eizenberg ., "The language of the prairie: Frank Lloyd Wright's prairie houses", Environment and Planning B: Planning and Design, 8**,** pp: 295-323, 1981.

H. Liew., “Extending Shape Grammars with Descriptors”, Design Computing and Cognition (DCC) Conference Proceedings, pp: 417-436. 2004.

H. Lipson, J. B. Pollack., "Automatic design and Manufacture of Robotic Lifeforms", Nature 406, pp. 974-978, 2000

H. Stoeckle, J. Grundy, J Hosking.,”A framework for visual notation exchange”, Journal of Visual Languages and Computing, vol. 16, p 187-212, 2005.

H. Suzuki., A. Hidetoshi., F. Kimura., “Geometric Constraints and Reasoning for Geometrical CAD Systems”, Computers and Graphics, 14(2), pp. 211-24, 1990.

J. Cagan., “Engineering Shape Grammars: Where Have We Been and Where are We Going?”, in E. K. Antonsson, J. Cagan, eds., Formal Engineering Design Synthesis, Cambridge University Press, Cambridge, UK, 2001.

J. D. Kiper, E. Howard, C. Amesa., “Criteria for Evaluation of Visual Programming Languages”, Journal of Visual Languages and Computing, Vol. 8, p175-192, 1997.

J. D. M. Andries, G. Angles., “A Hybrid Language for an Extended Entity - Relationship Language”, Journal for Visual Languages and Computing (7), p 321 – 352, 1996.

J. D. Summers, N. Vargas-Hernandez, Z. Zhao, J. Shah, Z. Lacroix., “Comparative Study of Representation Structures for Modeling Function and Behavior of Mechanical Devices”, Proceedings of the DETC-2001, ASME, CIE-21243, Pittsburgh, PA, 2001,.

J. D. Summers, J. Shah., “Empirical Studies for Evaluation and Investigation of a New Knowledge Representation Structure in Design Automation”, Proceedings of the DETC-2002, ASME, CIE-34488, Montreal, Canada, 2002.

J. D. Summers, J. Shah., "Developing Measures of Complexity for Engineering Design", Design Engineering Technical Conferences, DETC-2003, Chicago, IL, DTM-48633, 2003.

J. D.Summers, J. Shah,B. Bettig., "The Design Exemplar: A New Data Structure for Embodiment Design Automation", Journal of Mechanical Design, 126 (5): 775-87, (2004).

J. D. Summers., "Development of a Domain and Solver Independent Method for Development of Mechanical Engineering Embodiment Design", Arizona State University, 2004.

J. D. Summers, J. Shah., "Exemplar Networks: Extensions of the Design Exemplar", Design Engineering Technical Conferences, Computers in Engineering, DETC-2004, Salt Lake City, UT, CIE-57786, (2004).

J. D. Ullman., “Fundamental Concepts of Programming Systems”, Addison-Wesley, Reading MA, 1976.

J. D. Ullman., “Principles of Database and Knowledge-Base Systems: The New Technologies”, Principles of Computer Science Series, 14”, W.H. Freeman & Company, 1989.

J. Gips, G. Stiny., “Production Systems and Grammars: A Uniform Characterization”,Environment and Planning B, 7, pp: 399-408, 1980.

J. Heisserman, R. Woodbury., “Generating Languages of Solid Models”, p103 – 112, 2nd ACM Solid Modeling, Montreal, Canada, 1993.

J. Jaffar, M. Maher., "Constraint logic programming: a survey," Journal of Logic Programming, vol. 19-20, pp. 503 - 582, 1994.

J. S. Gero., “Creativity, emergence and evolution in design”, Knowledge.-Based Systems, Vol. 9(7): pp: 435-448, 1996.

J. S. Gero., ”Conceptual designing as a sequence of situated acts”, AI in Structural Engineering, pp: 165-177, 1998.

J. S. Gero., ”Representation and Reasoning about Shapes”, Cognitive and Computational Studies in Visual Reasoning in Design, COSIT, pp: 315-330, 1999.

J. Thoma., “Introduction to Bond Graphs and their Applications”, Pergamon Press, New York, NY. 1975.

J. Thorpe, J., Mechanical System Components, Allyn and Bacon, Boston, MA, 1989.

K. D. Forbus, P. Nielsen, B. Faltings., "Qualitative spatial reasoning: The clock project", Artificial Intelligence, 51, pp: 417-471, 1991.

K. Jemielniak., " Web: http://www.cim.pw.edu.pl/labview/en/ ",2004

K. Wittenburg, L. Weitzman., "Relational grammars: theory and practice in a visual language interface for process modeling", AVI'96, Gubbio, Italy, 1996.

K. N. Whitley., "Visual Programming Languages and the Empirical Evidence For and Against", Journal of Visual Languages and Computing, Vol. 8, p: 109-142, 1997.

LabVIEW., "Web: http://www.ni.com/devzone/lvzone/dr_vi_archived6.htm", National Instruments Corporation, 2006.

L. Fegaras., "Web: http://lambda.uta.edu/cse5317/notes/",University of Texas at Arlington, 2005

L.R. Nackman, M. A. Lavin, R. H. Taylor, W. C. Diretrich, Jr., D. D. Grossmann., "AML/X: A Programming Language for Design and Manufacturing", ACM Fall joint computer conference, p 145 – 159, Dallas, Texas, 1986.

M. Agarwal, J. Cagan., "On the use of Shape Grammars as Expert Systems for Geometry based Engineering Design", Artificial Intelligence in Engineering Design, Analysis and Manufacturing, Vol. 14, pp: 431-439, 2000.

M. Bernini, M. Mosconi., "VIPERS: a Data Flow Visual Programming Environment Based on, the Tel Language", Proceedings of the workshop on Advanced visual interfaces, p: 243 – 245, 1994.

M. M. Burnett, M. J. Baker., "A Classification System of Visual Programming Language", Journal of Visual Language and Computing,1994.

Merriam-Webster, Incorporated., "Web: http://www.m-w.com/cgi-bin/dictionary?book=Dictionary&va=iconTeksoft., "Web: http://www.teksoft.com/procam/procam2.htm", Teksoft Software Solutions, 2006.

M. Erwig., "Visual type inference", Journal of Visual Languages and Computing & Computing, Vol. 17, p: 161 – 186, 2006.

M. Jamshidi, H. R. Parsaei., "Design and Implementation of Intelligent Manufacturing Systems",Prentice Hall PTR; Facsimile edition, 1995.

M. P. Groover., E. W. Zimmers, Jr., "CAD/CAM: Computer-Aided Design and Manufacturing",Pearson Education , Inc., Delhi, 2003

M. Suwa., B. Tversky., "What Do Architects and Students Perceive in their Design Sketches? A Protocol Analysis", Design Studies, 18(4), pp. 385-403, 1997.

N. C. Shu., "Visual Programming", Van Nostrand Reinhold Company, New York, 1988.

N. Drakos, "Web: http://www.msri.org/about/sgp/jim/software/vps/desc/node2.html", Computer Based Learning Unit, University of Leeds, 1996

O. Banyasad, P. T. Cox., "Solving design problems in a logic-based visual design environment", Technical Report CS 2001-04, Dalhousie University, Canada, 2001.

O. Banyasad, P. T. Cox., "Implementing Lograph", Technical Report CS 2001-05, Dalhousie University, Canada, 2001.

O. Banyasad, P. T. Cox., “On Translating Geometric Solids to Functional Expressions”, PPDP, Uppsala, Sweden, 2003.

O. Banyasad, P. T. Cox., “Integrating design synthesis and assembly of structured objects in a visual design language”, Technical Report CS 2001-05, 2001, Dalhousie University, Canada.

P. Deak, C. Reed, G. Rowe., ”Grammars For Spatial Design Modelling”, School of Computing, University of Dundee, UK, 2006.

P. H. Winston., “Artificial Intelligence”.,Addison- Wesley Publishing Company., 1993.

PowerShape., “Web :http://www.moldmaking-cadcam.com/”, Delcam plc, 2006.

P. T. Cox, T. J. Smedley., "A Declarative Language for the Design of Structures,", IEEE Symposium on Visual Languages (VL '97), p. 438, 1997.P. T. Cox, T. J.Smedley., “LSD: A Logic-Based Visual Language for Designing Structured Objects”., Journal of Visual Language and Computing, v9, Academic Press (1998), 509-534, 1998.

P. V. Zee, M. M. Burnett, M. Chesire., "Retire Superman: Handling Exceptions Seamlessly in a Declarative Visual Programming Language," IEEE Symposium on Visual Languages, Boulder, Colorado, pp. 222-230, 1996.

R. Ahmad., “Visual Languages: A New Way of Programming”, Malaysian Journal of Computer Science, Vol. 12, pp: 76-81., June 1999.

R. Hartley, H. Pfeiffer, "Visual Representation of Procedural Knowledge," p: 63, 2000 IEEE International Symposium on Visual Languages (VL'00), 2000.

R. Sedgewick, R., "Algorithms in C++", Addison-Wesley, Menlo Park, CA, 1998.

R. Sedgewick,P. Flajolet., "An Introduction to Analysis of Algorithms", Addison-Wesley, Menlo Park, CA, R. W. Djang, M. M. Burnett, R. D. Chen., "Static Type Inference for a First-Order Declarative Visual Programming Language with Inheritance", Journal of Visual Languages and Computing, Vol. 11., p 191-235, 2000.

S. Anandan, J. D. Summers., “Similarity Metrics Applied to Graph Based Design Model Authoring”, Computer Aided Design & Applications, 2006

S. Hansche., Official (ISC)2 Guide to the CISSP Exam, CRC Press, Boca Raton, FL, 2003.

Simulink., “User’s Guide”, “Web: http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/”, The Mathworks, Inc., 2006.

S. J. Shin., “The Iconic Logic of Peirce’s Graphs”, MIT Press, Cambridge, MA, 2002.

S, K. Chang., “Principles of Visual Programming Systems”, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

S. K. Chang, M. M. Burnett, S. Levialdi, K. Marriott, J. J. Pfeiffer, S. L. Tanimoto., "The Future of Visual Languages," vl, p. 58, IEEE Symposium on Visual Languages, 1999.

S. K. Lee, K. Y. Whang., “VOQL: A Visual Object Query Language Inductively Defined Formal Semantics”, Journal of Visual Languages and Computing, Vol. 12, p: 413-433, 2001.

SolidWorks., “Web: http://www.solidworks.com/index.html”, SolidWorks Corporation, 2006.

S. Venkatamaran., “Integration of Design by Features and Feature Recognition”, Master’s Thesis, Arizona State University, Tempe, AZ, 2000.

T. F. Stahovich., “Artificial Intelligence for Design”, in Formal Engineering Design Synthesis, Chapter 7, E. K. Antonsson and J. Cagan, eds., Cambridge University Press, Cambridge, pp: 228-269, 2001.

T. Green., "Noddy's Guide to Visual Programming", Interfaces, Autumn, 1995.

T. Ichikawa, M. Hirakawa., "Visual Programming--Toward Realization of User-Friendly Programming Environments", Exploring Technology: Today and Tomorrow--Proceedings of the Fall Joint Computer Conference, IEEE Computer Society, pp. 129-137, Dallas, 1987.

T. Kamada, S. Kawai., “A General Framework for Visualizing Abstract Objects and Relation”, ACM Transactions on Graphics, Vol. 10. p: 1 - 39, 1991.

T. Laakko, M. Miintyla., “A feature definition language for bridging solids and features”, 2ND ACM Solid Modelling, Montreal, Canada, 1993.

T. Mccabe, C. Butler., "Design Complexity Measurement and Testing", Communications of the ACM, 31 (12): 1415-25, 1989.

W. D. Engelke., “How to Integrate Cad/cam Systems: Management and Technology” Marcel Dekker, 1987.

Y. Ota, "Historical Role and Capability of Visual Language," vl, p. 308 - 312, IEEE Symposium on Visual Languages, 1999.

Y. Hu., “Optimimal design for additive manufacturing of heterogeneous Objects”, PhD thesis, Clemson Univesity, 2005.