Emergence Through Conflict
The Multi-Disciplinary Design System (MDDS)

By
Anas Alfaris

Master of Science Candidate in Computation for Design and Optimization
Massachusetts Institute of Technology, 2009

Master of Science in Architecture Studies
University of Pennsylvania, 2002

Master of Architecture and Building Technology
University of Pennsylvania, 2000

Bachelor of Science in Architecture and Building Engineering
King Saud University, 1998

Submitted to the Department of Architecture in the Partial Fulfillment of the Requirement for the degree of

Doctor of Philosophy in Architecture, Design and Computation

At the

Massachusetts Institute of Technology

June 2009

© Anas Alfaris All rights reserved

The author herby grants MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

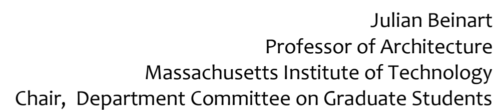
Signature of author

Department of Architecture, MIT
May 1st, 2009

Certified by:

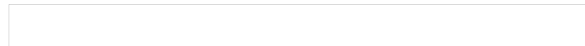
William Mitchell
Professor of Architecture and Media Arts and Sciences
Massachusetts Institute of Technology
Dissertation Advisor

Accepted by:



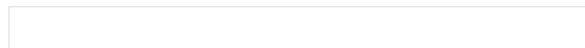
Julian Beinart
Professor of Architecture
Massachusetts Institute of Technology
Chair, Department Committee on Graduate Students

Dissertation Committee

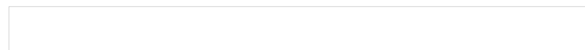


William Mitchell
Professor of Architecture and Media Arts and Sciences
Department of Architecture, MIT
Dissertation Advisor

Olivier de Weck
Associate Professor of Aeronautics and Astronautics
and Engineering Systems
Department of Aeronautics and Astronautics
and Engineering Systems Division, MIT



Terry Knight
Professor of Design and Computation
Department of Architecture, MIT



John Williams
Associate Professor of Information Engineering, Civil and Environmental
Engineering, and Engineering Systems
Department of Civil and Environmental Engineering
and Engineering Systems Division, MIT

Emergence Through Conflict: The Multi-Disciplinary Design System (MDDS)

By Anas Alfaris

Submitted to the Department of Architecture on May 1st, 2009
in the Partial Fulfillment of the Requirement for the Degree of
Doctor of Philosophy in Architecture, Design and Computation

Abstract

This dissertation proposes a framework and a group of systematic methodologies to construct a computational Multi-Disciplinary Design System (MDDS) that can support the design of complex systems within a variety of domains. The way in which the resulting design system is constructed, and the capabilities it brings to bare, are totally different from the methods used in traditional sequential design.

The MDDS embraces diverse areas of research that include design science, systems theory, artificial intelligence, design synthesis and generative algorithms, mathematical modeling and disciplinary analyses, optimization theory, data management and model integration, and experimental design among many others.

There are five phases to generate the MDDS. These phases involve decomposition, formulation, modeling, integration, and exploration. These phases are not carried out in a sequential manner, but rather in a continuous move back and forth between the different phases.

The process of building the MDDS begins with a top-down decomposition of a design concept. The design, seen as an object, is decomposed into its components and aspects, while the design, seen as a process, is decomposed into developmental levels and design activities. Then based on the process decomposition, the architecture of the MDDS is formulated into hierarchical levels each of which comprises a group of design cycles that include design modules at different degrees of abstraction. Based on the design object decomposition, the design activities which include synthesis, analysis, evaluation and optimization are modeled within the design modules. Subsequently through a bottom-up approach, the design modules are integrated into a data flow network. This network forms MDDS as an integrated system that acts as a holistic structured functional unit that explores the design space in search of satisfactory solutions.

The MDDS emergent properties are not detectable through the properties and behaviors of its parts, and can only be enucleated through a holistic approach. The MDDS is an adaptable system that is continuously dependent on, and responsive to, the uncertainties of the design process. The evolving MDDS is thus characterized a multi-level, multi-module, multi-variable and multi-resolution system.

Although the MDDS framework is intended to be domain-independent, several MDDS prototypes were developed within this dissertation to generate exploratory building designs.

Thesis Advisor: William Mitchell
Title: Professor of Architecture and Media Arts and Sciences

Acknowledgment

This was the last part of my thesis, and I found myself writing it the night before submitting my final copy. Nevertheless, it was one of the hardest parts to write, perhaps because it includes so much of what my Ph.D. journey was all about.

I remember when I first sat down with Professor Fahad Alsaeed at KFUPM to discuss my interest in earning a Ph.D. He explained to me that it would be a long but rewarding journey that would eventually provide me with a skill set for overcoming important problems. Yet, when I started this journey I did not realize that my Ph.D. lay beyond several years of hard work, three Master degrees, over fifty courses, qualifying exams, several presentations and committee meetings.

Now that I have come to the end of my journey, I realize that several professors, collaborators, friends, institutions, and family members have helped me reach my final destination. However, acknowledging everyone who has helped me along the way would be an impossible task.

I would first like to thank my advisor, Professor William Mitchell, for sharing his life of experience and for his inspiring intellectual leadership. He has exceeded every positive expectation I had for my advisor. I cannot imagine a better mentor for my personality and interests. He never forced an agenda on my research but instead contributed important guidance to help me pursue my own path. In spite of the pressure to be confined within a well-defined discipline for the sake of developing deep understanding, Bill taught me that engineering design has no boundaries and that depth can also exist across disciplines. From him, I learned how to tackle problems with an open mind and not to fear new and unstructured problems but to seek them. Bill has also opened wide domains of research for me, whether at the Media Lab or Design Lab, and for that I will always be grateful.

I was also fortunate to have three additional faculty members on my committee who all think outside the normal bounds of traditional disciplines: Professors Oliver de Weck, Terry Knight and John Williams. Each member was selected for my doctoral committee because of their perspectives on joining design with computation.

I cannot express in words what it has meant to me to have had the opportunity to work with Professor Oliver de Weck. His exceptional experience with multi-disciplinary optimization and system engineering and architecture had a strong influence on my work and research. Oli provided

me with valuable advice throughout this academic endeavor and always believed in my ideas and my work. I also cannot thank him enough for providing me with amazing research opportunities whether at the System Engineering Division or the Strategic Engineering Group.

Professor Terry Knight has inspired my interest in generative synthesis models. She was incredibly influential in guiding my intellectual growth and has provided invaluable insights and contributions. She always encouraged me to recognize the value of my work and showed me how to view the world from new perspectives.

Although I previously believed I was a decent programmer, I believe now that the moment I started working with professor Williams was the first time I truly learned to program. John brought to this research precious contributions in relation to information technology and systems that drew from his extensive knowledge of the field and theoretical background, and for that I am deeply thankful.

I would especially like to acknowledge Professor George Stiny. Though not a member of my committee, he provided invaluable feedback and support, both academic and personal, throughout my doctoral journey. I would also like to thank Professors John Fernandez and John de Manchaux who were both influential in my joining MIT.

I have enjoyed my time at MIT tremendously and had the chance to meet a collection of great minds. I was lucky to find many homes at MIT in different programs and departments. I would like to thank great faculty, colleagues, and friends in the Department of Architecture, the Media Lab, CSAIL, and the Engineering Systems Division (ESD), the School of Engineering, Design Lab, the Strategic Engineering Group, the Design and Computation Group and the Computation for Design and Optimization Program.

So many people contributed to this thesis, knowingly and unknowingly. From those I would like to acknowledge Riccardo Merello, a great friend and collaborator to whom I owe a huge debt of gratitude. He dedicated time and effort to the maturation of my ideas toward practical thinking. The experiments in the last section of this thesis would not have been possible if not for his insight and contributions. I will always remember our long hours online discussing different parts of those experiments. I would also like to thank Maher Elkhaldi for being a great friend at MIT and beyond. Our discussions and his help in the synthesis models chapter were invaluable. He was always there when I needed him, and he has always been patient with my short temper, and for that I cannot thank him enough. I would also like to acknowledge Saud Sharaf for our many long and useful informal discussions about the thesis and what the future may hold for us.

In addition, I learned a great deal from many friends at MIT. From this group of great future educators, I would like to mention Dan Iancu for helping me understand what optimization theory is all about. I would also like to acknowledge Harn Wei Kua for being a great friend and taking time from his busy schedule to teach me all about his combined passion for sports and

physics. Furthermore, I would like to thank Alexandre Marques, who, with his superb math skills, helped me a great deal in solving difficult problem sets.

In addition to the individuals I have already mentioned, I would like to thank other collaborators who have helped me in several research projects including Kenneth Namkung, Alexandros Tsamis, Ryan Chin, Meredith Elbaum, Philip Gayer, and Nii Armar among others. I would also like to mention other friends and colleagues who have been very supportive throughout my Ph.D., including Yanni Loukissas, Axel Killian, Dennis Shelden, Saeed Arida, Kshitij Prakash, Kenfield Griffith, Sergio Araya, Franco Vairani, Daniel Cardoso, Lira Nikolovska, Carlos Barrios-Hernandez, Neri Oxman, and Kaustuv De Biswas among others.

I would also like to thank the amazing staff at MIT for providing a sense of community. I would especially like to thank Renee Caso for always keeping me on track and on schedule and for her amazing support throughout my stay at MIT. I would also like to mention Cynthia Wilkes for helping me navigate Professor Mitchell's busy schedule for several years. In addition, I would like to acknowledge Tom Fitzgerald for his superb IT support and for always being there when I needed him. I would also like to thank Eric Markowsky who was kind enough to proofread this thesis on very short notice.

I should also thank a few friends outside the MIT community whose help and support have been instrumental in completing this journey. I would like to thank Ahmed Rashad for being a close and loyal friend throughout the last few years. I have no doubt that I could not have completed this thesis without his help in formatting it, and this is only one of the many things he has helped me with over the years. I would also like to acknowledge my dear friend for fifteen years, Bandar Alkahlan, who stayed up with me two nights in a row to help me finalize my dissertation defense presentation. I would like to thank my friend, Sherif Abdelmohsen, without whose help in editing several chapters of this thesis, I could not have finished on time.

Furthermore, I would like to thank Professors Peter McCleary, Ali Malkawi Branko Kolarevic, and William Braham from the University of Pennsylvania for setting me on track early on in this journey. I should also thank my uncle Professor Abdullah Alrawaf for his continuous support and encouragement throughout my academic career.

I would also like to thank Phoenix Integration for being so helpful in providing me with ModelCenter software and for their excellent technical support, especially Chad Kasell and J Simmons who have been extremely helpful and patient with my endless inquiries.

I would also like to thank the Saudi Ministry of Higher Education for making this thesis possible by funding and supporting this research project, and I would like to thank Ambassador Hamad Alfaris for initially helping me embark on this journey which would not have happened without his unwavering support.

I have received enormous support from my family; without it this thesis would never have been completed. I would like to thank my dear brother Eyas, who almost threw me off track on this research, but was also very supportive in so many ways. I would also like to thank my lovely sister Nasreen for her endless support and encouragement and my youngest sister Sawsan for always being there for me.

I want to thank my dear wife Manal Alaamery for sharing this journey with me and for the wonderful time we spent in Boston. Although burdened by her own Ph.D. and fellowship, she managed to find time to support me through all of my ups and downs, though I am sure she felt the downside more often due to my inclination to complain more than praise. Manal, thank you for your patience and support during this truly difficult experience. I would also like to mention my wife's parents who were very supportive as well.

To my dear son Faris and daughter Aseel, I ask your forgiveness, for you have been extraordinarily patient throughout this journey although you have suffered the most from it. Unfortunately, I know I cannot recreate the moments we lost, but I will try my best to make up for them. Thank you for being part of my life.

This work is dedicated to my parents for their unwavering support and enthusiasm for my academic adventures. I know it has been a long journey, but it is finally over (at least this degree). I would like to thank my dad, architect Faris Alfaris, for always guiding me through life with his clarity of view, wisdom and his uncompromising honesty. I would also like to thank my mom, Dr. Haya Alrawaf, for her infinite prayers and encouragement. You have taught me that learning is a life-long adventure worth pursuing. Although you can call me Doctor now, I will forever be a student and I don't intend ever to stop learning.

Finally, as is clearly obvious in this acknowledgment, this was a shared experience and therefore I would like to thank everyone who had a part in this and to everyone who has touched my life. You should know who you are, and you are all in my thoughts.

Anas Alfaris

Table of Contents

Emergence Through Conflict:
The Multi-Disciplinary Design System (MDDS)

1. Introduction

1.1. Motivation 17
1.2. Challenges in Computational Design Systems 18
1.3. Research Methodology 23
1.4. Thesis Structure 29

2. Theoretical Background

2.1. Design Science 33
2.1.1 Design as an Object and a Process 34
2.1.2 Design Process Domains 36
2.1.3 Design Process Evolution 40
2.1.4 Design Process Activities 42
2.2. Systems Theory 43
2.2.1 System Concepts 43
2.2.2 System Architecture 47
2.2.2.1 Form and Function 49
2.2.2.2 Architecture of Integration and modularity 51
2.2.2.3 System Structure 54
2.2.2.4 Behavior 62
2.2.2.5 Process of Creating Architecture 62

3. Decomposition

3.1 What is Decomposition? 65
3.2 Design Object Decomposition 67
3.2.1 Component/Physical Decomposition 68
3.2.2 Design Aspects Decomposition 69
3.3 Design Process Decomposition 69
3.3.1. Design Development Decomposition 69
3.3.1.1. Design Development Models 70
3.3.1.2. Proposed Design Development Model 75
3.3.2. Design Activity Decomposition 75
3.3.2.1. Design Activity Models 76
3.3.2.2. Proposed Activity Decomposition Model 85
3.3.3. Hybrid Design Process Models 85
3.3.4. Decomposition and Design Views 88

4. Formulation

4.1 What is Formulation? 91
4.2 Process Analysis and Structuring 92
4.3. Iteration and Coupling 97
4.4 Process and Formulation Modeling 99
4.4.1. Network Models 100
4.4.1.1. Data Flow Diagrams 100

4.4.1.2. Functional Flow Block Diagrams 101
4.4.2. Formulation Modeling Languages 105
4.4.2.1. Unified Modeling Language 105
4.4.2.2. System Modeling Language 112

5. Modeling

5.1 What is a Model? 117
5.2.The Mathematical Model 118
5.2.1 Elements of Mathematical Models 119
5.2.2 Constructing Mathematical Models 121
5.2.3 Types of Mathematical Models in Design 124
5.3. Synthesis Models 127
5.3.1 What is a Synthesis Model? 127
5.3.2 The Synthesis Model Structure 128
5.3.2.1 Algorithms in design 128
5.3.2.2 Parameters 129
5.3.2.3 Design Relationships 132
5.3.2.4 Formal Grammars 135
5.3.3 Computational Representation of Synthesis Models 149
5.3.4 Modeling Variation 155
5.3.4.1 Synthesis Design Vector 155
5.3.4.2 Solution Space 156
5.3.4.3 Knowledge and Performance Encoding 160
5.4 Analysis Models 163
5.4.1 What is an Analysis Model? 163
5.4.2 Model classifications based on the nature of model 164
5.4.2.1 Qualitative and Quantitative Models 165
5.4.2.2 Continuous and Discrete Models 166
5.4.2.3 Deterministic and Stochastic Models 166
5.4.2.4 Static and Dynamic Models 167
5.4.2.5 Linear and Nonlinear Models 168
5.4.3. Analysis Algorithms 168
5.4.3.1 Theoretical Models 170
5.4.3.1.1 Analytical Models 170
5.4.3.1.2 Numerical Models 172
5.4.3.2 Approximation Techniques 183
5.5 Evaluation Models 192
5.5.1. What is an Evaluation Model? 192
5.5.2 Single and Multi Objective Evaluation and Optimization 193
5.5.3 Multiobjective Methods 195
5.5.3.1 Decision Making before Search 197
5.5.3.1.1 Method of Weighted-Objectives 197
5.5.3.1.2 Utility 199
5.5.3.2 Search Before decision making 200
5.5.3.2.1 MOGA 203
5.6 Optimization Models 205
5.6.1 What is an Optimization Model? 205
5.6.2 Mathematical Formulation. 206
5.6.3 Classification of Optimization Problems 207

5.6.3.1 Based on Constraints 208
5.6.3.2 Based on Design Variables 209
5.6.3.3 Based on nature of objective function 209
5.6.4 Classification of Optimization Algorithms 211
5.6.4.1 Deterministic Algorithms 212
5.6.4.1.1 Derivative-Free Methods 212
5.6.4.1.2 Gradient Methods 218
5.6.4.2 Heuristic Algorithms 223
5.6.4.2.1 Evolutionary Algorithms 223
5.6.4.2.2 Simulated Annealing 229
5.6.4.2.3 Tabu Search 231

6. Integration

6.1 What is Integration? 233
6.2 Interface Design 234
6.3 Module Integration Modes 237
6.3.1 Middleware 238
6.3.1.1 Encapsulation (Wrappers) 241
6.3.1.2 Web Services 243
6.3.1.2.1 Extensible Markup Language (XML) 244
6.3.1.2.2 SOAP 245
6.3.2 Integrated Computing Environments 246
6.3.2.1 The One-Software Approach 247
6.3.2.2 Problem Solving Environments 247

7. Exploration

7.1 What is Exploration? 253
7.2 Pre-Search 254
7.2.1 Parameter Studies 254
7.2.2 Design of Experiments 256
7.2.2.1 Factors, Levels and Responses 257
7.2.2.2 Treatments 258
7.2.2.3 Effects 258
7.2.2.4 Full Factorial 259
7.2.2.5 Fractional Factorial Design 260
7.2.3 Latin Hypercubes 261
7.2.4 Orthogonal Arrays 263
7.3 Post- Search 264
7.3.1 Sensitivity Analysis 264

8. The Multi-Disciplinary Design System (MDDS)

8.1 What is MDDS? 265
8.2 MDDS Framework 266
8.2.1 Decomposition 268
8.2.2 Formulation 270
8.2.3 Modeling 274
8.2.3.1 Synthesis 275
8.2.3.2 Analysis 277

8.2.3.3 Evaluation 277
8.2.3.4 Optimization 279
8.2.4 Integration 281
8.2.5 Exploration 283
8.3 System Evolution 285
8.3.1 Complexity 285
8.3.1.1 Multi-Level 286
8.3.1.2 Multi-Module 287
8.3.1.3 Multi-Variable 287
8.3.1.4 Multi-Resolution 288
8.3.1.5 Decoupling 289
8.3.2 Adaptability 290
8.3.3 Optimality 292
8.3.4 Time 293
8.4 System Behavior 295
8.4.1 Performance Driven Design 295
8.4.2 Collaborative Multidisciplinary Perspective 297
8.4.3 Emergence 298
8.5 MDDS Team and Environment 300
8.5.1 MDDS Team 300
8.5.2 MDDS Environment 303

9. Experiments
9.1. Experiment1 | Level 1 305
9.1.1 Concept 305
9.1.2 Decomposition 306
9.1.2.1 Component Decomposition 306
9.1.2.2 Aspect Decomposition 306
9.1.2.3. Development Decomposition 307
9.1.2.4. Activity Decomposition 307
9.1.3 Formulation 308
9.1.4 Modeling 310
9.1.4.1 Synthesis Modules 310
9.1.4.2 Analysis Modules 314
9.1.4.3 Evaluation Modules 322
9.1.4.4 Optimization Modules 324
9.1.5 Integration & Exploration 327
9.2 Experiment 1 | Level 2 330
9.2.1 Design Concept 330
9.2.2 Decomposition 332
9.2.2.1 Component Decomposition 332
9.2.2.2 Aspect Decomposition 332
9.2.2.3 Development Decomposition 333
9.2.2.4 Activity Decomposition 333
9.2.3 Formulation 334
9.2.4 Modeling 336
9.2.4.1 Synthesis 336
9.2.4.2 Analysis 339
9.2.4.3 Evaluation 342
9.2.4.4 Optimization 344
9.2.5 Integration 344

9.2.6 Exploration 346
9.2.6.1 Experiment 1 346
9.2.6.2 Experiment 2 350
9.2.6.3 Experiment 3 353
9.3 Experiment 2 | Level 1 357
9.3.1 Design Concept 357
9.3.2 Decomposition 357
9.3.2.1 Component Decomposition 357
9.3.2.2 Aspect Decomposition 358
9.2.2.3 Development Decomposition 358
9.3.2.4 Activity Decomposition 359
9.3.3 Formulation 360
9.3.4 Modeling 362
9.3.4.1 Synthesis 362
9.3.4.2 Analysis 364
9.3.4.3 Evaluation 368
9.3.4.4 Optimization 369

10. Conclusion

10.1 Thesis Summary 383
10.1.1 Decomposition 383
10.1.2 Formulation 383
10.1.3 Modeling 384
10.1.4 Integration 388
10.1.5 Exploration 389
10.2 Thesis Contributions 389
10.2.1 A Computational Design System Model 390
10.2.2 A Multidisciplinary Design System Model 391
10.2.3 An Evolutionary Design System Model 391
10.2.4 An Adaptable Design System Model 393
10.2.5 A Generative Performance-Driven Design System Model 394
10.2.6 A Design System Model with Emergent Behaviors 394
10.2.7 A Model that Reduces Design Iteration Time 395
10.2.8 A Model that Redefines the Design team and Studio 396
10.2.9 A Design System for Integral and Modular Architectures 396
10.2.10 A New Approach to Building Civic Architecture 397
10.3. Limitations and Difficulties 397
10.3.1 Synthesis Complexity 397
10.3.2 Analysis Representation 397
10.3.3 Multi-Level Optimality 398
10.3.4 Evaluation Visualization 398
10.3.5 Algorithmic Exploration 398
10.3.6 Setup Time 399

Figures List 401

Tables List 412

Bibliography 413

1. Introduction

1.1. Motivation

A design process is highly dependent on the available tools, and at the same time, the design processes and tools have a strong impact on the designed artifact. Both computer users and non-users share the belief that computers have had, and will continue to have, a significant and deep influence on design; whether that influence is desirable or not is debatable.

We know that computers can be used in a diligent manner that enables humans to surpass their physical limitations. Computational design systems have already been developed for the purposes of automation and design assistance. These systems free the designer from various concerns about effort, labor or complexity.

Descartes and Kant discuss issues relevant to the potential of machines to exceed human limitations. Descartes introduces a significant and intriguing position. Descartes' question "how can a designer build a device which outperforms the designer's specifications?" (Cariani, 1991). Kant on the other hand inquired, "How can work full of design build itself up without a design and without a builder?" (Jaki, 1981).

Clearly the ability of a computational design system in releasing the designer from various concerns owes basically to the power of the human mind, which can invent devices that can exceed its own limitations. However, can such actions be considered intelligent behavior?

The dilemma of intelligent behavior in design systems is that such intelligence is often not understood enough in the first place. Such a dilemma might be resolved by either augmenting our understanding or by producing systems that do not only automate design processes but as Descartes stated go beyond the specifications given to them (Cariani, 1991). The latter is the approach pursued in this thesis.

1.2 Challenges in Computational Design Systems

Design involves solving what Herbert Simon terms an ill-structured problem (Simon, 1973). An ill-structured problem is one that cannot be solved by a linear chain of reasoning derived from the problem statement. Furthermore, it might not have a unique solution but a multiplicity of solutions. These design problem characteristics imply the need for many assumptions within the design process that can only be verified after a solution is reached. Given the numerous inputs that feed into design, it is not surprising that design presents a technical challenge even for relatively well understood products (Eppinger and Gebala, 1991). The problem is further complicated in multi-disciplinary design by the need to satisfy each discipline's performance criteria. This makes computational design systems a difficult area of study, where creating effective systems requires the development of new ways to represent designs and evaluate their different disciplines' performance criteria collectively.

Recent possibilities facilitated by advancements in computational power and new developments in computer-based modeling and analysis methods, such as finite element analysis (FEA), computational fluid dynamics (CFD), visualization, process simulation and others, have enabled the simulation of design performance in virtual environments. This provided designers and engineers with information that can assist in some decision making (Paydarfar, 2001). However, these design models are usually discipline specific and hence lack the ability to supply sufficient understanding of the possible tradeoffs between different disciplines. Therefore the design insight gained through these tools and technologies remains limited and their potential to enhance and inform the design process has not been fully realized.

Design is a complex activity requiring knowledge spanning many different domains. Even the most rudimentary design activities demand scientific knowledge, engineering skills and artistic creativity. Many real-world design problems cannot be modeled by one single model. Systems like aircrafts, automobiles or skyscrapers consist of continually and mutually interacting subsystems. The behavior of such complex systems and products is controlled by a variety of physical phenomena that are analyzed by different disciplines and that interact at the same time. Therefore, they require using groups of complementary tools and models that integrated together can describe the whole process (Yilmaz and Oren, 2004; Zeigler et al., 2000). However, there are often integration-related difficulties in multidisciplinary computational design, especially regarding information gathering, flow and

translation.

Given that computer-aided design tools produce more data than conventional methods, a large amount of information will accumulate during the design process. Furthermore, the complexity with which this information flows contributes significantly to the difficulty of computational design systems. Examples of such complexities include information circuits, in which the information flow within a design activity is circular. These information circuits are a consequence of design iteration in which decisions are revised due to incomplete or imperfect available information. It is thus important to reduce the time and effort that computational design systems require to integrate and coordinate information in order to complete design iteration (Eppinger and Gebala, 1991).

In addition, transferring the design and analysis results from one discipline to another group is not as straightforward as it may seem. Sometimes it is required to convert the results of one group or model into a form that others can use. This can take the form of simply translating the syntax of one program output into another program input but it also can get more complex, such as the interpolation from a finite element structures grid to an aerodynamic analysis mesh.

With the integration and automation of many discipline models, data management issues such as data architecture, data configuration control, data deletion, data translation, data quality, and data access, will present a challenge to the whole integration process. A lack of robust integration of design discipline models adds a programmatic cost that often reduces the sum of each model's individual benefits. Therefore, the effective management and exchange of information among different disciplines is necessary in order to realize expected cost, time and quality benefits. Achieving this requires better software technologies as well as more active planning, communication and synchronicity. This is particularly difficult for non-colocated models, tools and teams.

Currently many designers spend most of their time trying to manage design information rather than performing actual design activities. The amount of time spent in generating and evaluating alternatives using computational design models and tools forces many designers to use these methods for validating a selected alternative rather than exploring and quantitatively analyzing multiple alternatives and speculative scenarios. This leaves a wide range of the design space unexplored. This unexplored space often comprises better performing solutions than others ever previously considered (Shea

et al., 2005).

As mentioned earlier, design of complex systems and artifacts depends on diverse design and engineering domains; these design problems usually comprise conflicting objectives. This constitutes another design challenge. To overcome this there is a need to support the rapid generation and evaluation of design alternatives to provide satisfactory design space search.

In regards to the challenges mentioned above, some progress has been made in both the processes involved with multidisciplinary design as well as the development of promising technologies that support the design process.

In 1985 Dr. Siu Tong, a graduate from MIT working at General Electric Corporate Research and Development Center, and his team deployed a generic engineering design management and optimization tool called the "software robot". The purpose of this software shell was the automation and integration of simulation models to provide optimal designs with special focus on the design and analysis of aircraft engines. This was later developed into "Engineous" which automates the process of running simulation-based design systems (Tong, 2001; Hedberg, 2005). The main concept behind this tool was performing manual iterations which are typically undertaken by engineers in the design process.

This and many similar efforts were developed in the aerospace industry for multidisciplinary design analysis (MDA) in which mathematical analysis models and tools and their integration and automation were utilized leading to an enhanced understanding of the design performance.

Using centralized databases and associated management systems contributed to resolving complex interconnections among multiple models that exchange large amounts of data. Furthermore, more vigorous component management tools are applied to smaller interacting computation-intensive components and applications. These are driven by the recent "component ware" focus in the software industry, in which systems offer generic tools to manage interacting software components.

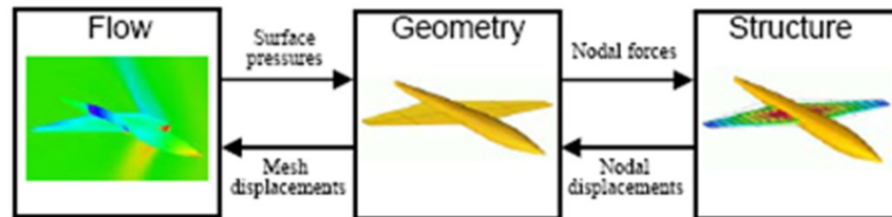
Based on the success in MDA, many attempts were made by the aerospace industry to link MDA with advanced optimization techniques and algorithms. This was driven by the need for strict integration between vehicle components in order to meet tough performance requirements. This necessity of integration in the context of tight performance coupling between system components

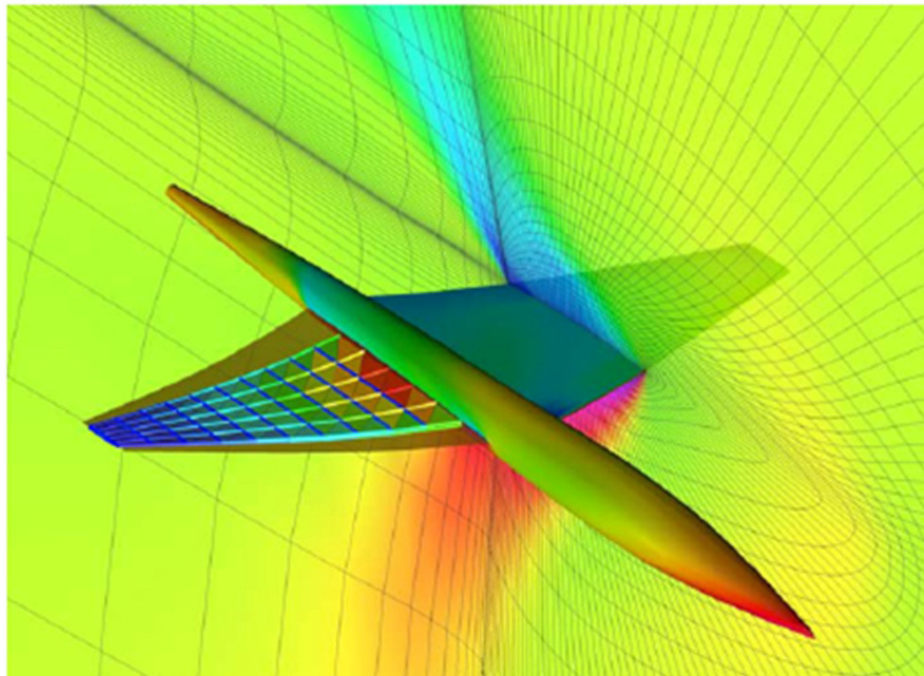
posed a real challenge for traditional design paradigms. The aerospace industry addressed these challenges by developing automated multidisciplinary design optimization (MDO) (Bowcutt et al., 2004).

MDO provides many powerful techniques for exploring very large design spaces and obtaining optimal solutions in large trade-space situations (McManus et al. 2004). MDO methods are especially useful in preliminary design activities which comprise multidisciplinary interactions and aim at achieving a design through rational trade-offs that satisfy constraints and at the same time maximize some objectives (Kroo, 1997a).

Figure 1.1:

Aircraft Design Optimization Framework Using MDO. Adopted from Martins, J. MDO Lab, University of Toronto





MDO in essence is a formalization of the design process that promotes careful and explicit problem formulation. This often creates a barrier to MDO application, but in general it can decrease the probability of costly redesign later on in product development. Figure 1.1.

MDO has been applied in many design problems. Examples of such applications were demonstrated in commercial transport aircraft research and design at Boeing and McDonnell-Douglas (Liebeck et al., 1996). This work resulted in the McDonnell-Douglas Blended Wing Body Concept (BWB), which constituted a non-traditional solution to the large subsonic transport problem. The BWB is an unstable tailless aircraft that has its passengers inside the center wing section and exploits boundary layer ingestion for improved fuel economy. In this design, many disciplines were tightly tied together, such as aerodynamics, structures, propulsion, stability, and control. This made the multidisciplinary analysis and design approach much more significant. Figure 1.2.

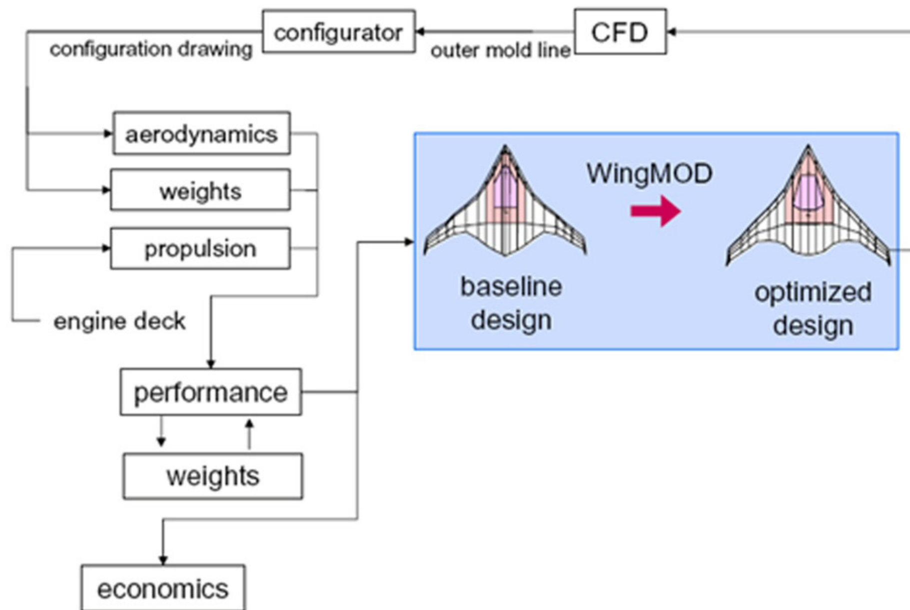


Figure 1.2:

MDO Framework for Blended Wing Body Concept (de Weck and Willcox, 2005).

The processes and technologies mentioned above have clearly advanced our understanding of computational design. They have attempted to address problems involving multi-disciplines as well as the use of several analysis models. They have also developed promising technologies that handle issues of integration, the management of several conflicting objectives as well as proposing methods for searching vast design spaces. However a few shortcomings still remain.

The design synthesis capabilities of such systems remain rudimentary even with the use of sophisticated CAD software. Although parametric models are powerful tools, capable of generating vast design spaces. Most of MDO application in literature demonstrates

simple dimensional parametric synthesis models. Within such examples the designer fixes topological variations of artifacts or elements, and the optimization merely varies dimensionality. Clearly better methods should be used for generating more varied design spaces. This will presumably enable multidisciplinary design teams to formally explore the performance of many more design alternatives, which should lead consequently to more novel designs and enhanced performance.

Another shortfall of current methods and technologies such as MDO is that they are restricted in their ability to evaluate the nature of evolving design requirements and how they might change during development and operation. Most work on MDO does not take into consideration the evolution of design complexity. Instead it deals with the problem of minimizing or maximizing a specified function with respect to a specified set of design parameters.

Design descriptions change as projects progress. Synthesis and analysis methods as well as constraints have to evolve consequently. A design cannot be described at the level required for manufacturing at the launch of the project. Due to this evolution, the complexity of both the design description of an artifact and the corresponding design models increase as design progresses.

Furthermore, the design technologies discussed focus on quantitative analysis and on optimization and searching for optimum solutions. However, these technologies do not take into account qualitative attributes of the design. An optimum solution quantitatively might not necessarily be the best solution. Better exploration of the design space might reveal solutions with better qualitative merits.

1.3 Research Methodology

In response to the challenges mentioned so far, the thesis will aim at developing a framework that employs computational design techniques that can improve the design and development of multidisciplinary complex systems and artifacts. To help develop a foundation and a theoretical discourse an investigation of theory, both of design science and methods and of systems theory, will be necessary. Later the framework for developing a multidisciplinary computational design system will be proposed. Along the way, knowledge from different domains will be sought to support the presented approach.

Many view design as a mysterious activity that does not lend itself to scientific examination, but that is not the case. Publications on

design methods date back to Roman times, particularly by Vitruvius (Gero, 1990). Design research continued and led to design thinking in the 19th century that articulated design as a process (Britt, 2000). In the 1960s major design research programs were also established. Further efforts led to information-processing models which were based on AI principles. These principles provided a momentum for renewed research into design in its various aspects (Simon, 1973; Coyne et al., 1990).

Designing is an activity that occurs with the prospect that the designed artifact will operate in both the natural and social worlds. Both these worlds introduce constraints on the variables and their associated values. As a result, design in this context can be viewed as a goal-oriented, constrained, decision making activity (Gero, 1990). Design can also be seen as an evolutionary process that evolves over time. Such a process can assist in handling design complexity by breaking the design into stages that move from the simple and abstract to the more complex and concrete.

Recently systems theory also provided a significant view on the process of system design, architecture and structure. It provides a framework for the description of several groups and objects that act in concert to produce some result. It investigates the principles common to all complex entities and the models which can be used to describe them. Both the computational software tools used for the purpose of design and the complex designed artifacts produced are considered systems. It follows that in order to better understand them, we need to understand systems and systems architecture.

As the title suggests, this thesis attempts to develop a Multi-Disciplinary Design System (MDDS) framework that would enhance the design of complex systems and artifacts through an efficient process. The framework that will be presented supports the design of multidisciplinary complex systems and artifacts within a variety of domains.

This framework would embrace and integrate diverse areas of research such as design science, systems theory, process modeling techniques, generative synthesis algorithms, multidisciplinary analysis, optimization theory, data management and integration, and design space exploration techniques. In addition the MDDS framework supports the exploitation of a group of emerging design computing technologies and software products to accomplish design and performance benefits.

The MDDS framework is a generic framework that suggests a group of systematic methodologies that eventually lead to a fully realized

and integrated design system. Within this system, complexities of the design should be handled and the uncertainty of its evolution can be managed. In addition, vast design spaces can be searched while solutions are intelligently modified, their performance evaluated, and their results aggregated into a compatible set of design decisions.

There are many stages however to generating the MDDS. These stages involve decomposition, formulation, modeling, integration, and exploration. These stages are not carried out in a sequential manner, but rather there is a continuous need to move back and forth to previous and subsequent stages. Figure 1.3.

After a design team identifies a design concept at a certain level which can best perform the design requirements, a top down decomposition of the concept is implemented by each discipline involved in the design. Complexity of a design can be managed or solved through partitioning it into smaller elements and observing each independently. There are different decomposition strategies that can be applied to the design artifact as well as to the design process.

Design process modeling then follows through formulation, providing an improved understanding of the process properties. Formulation enables the visualization of data and control flow. Different design processes and architectures can be compared and evaluated. MDDS architecture is broken down into hierarchical levels each of which comprises a group of cycles and modules at different degrees of abstraction. An iterative cycle between decomposition and formulation is required to reach an acceptable system architecture.

Mathematical models for each of the elementary modules are then developed. Each module represents a design activity that is formalized into computational models. These include synthesis, analysis evaluation and optimization modules which are later connected together in order to automate the process of design search.

One of the main challenges presented earlier is adequate design generation, also known traditionally as synthesis. Through several attempts to shorten design cycles and getting more robust solutions, design synthesis processes have been formalized. In the computer age, formal design methodologies, together with algorithmic descriptions, could be used to obtain automatic design synthesis while handling problems that might not be open to solution by the

unaided human mind.

Therefore, within the proposed framework the design concept will be decomposed into a set of synthesis models by extracting design intentions and formulating a collection of design variables, parameters, rules and algorithms. This mode of representation and formalism can be used within a computational environment to breed new design configurations.

Analysis models and simulations are then used to predict the behavior and performance of a specific design. An analysis model infers from a design solution characteristics that are relevant to a particular discipline. A design problem usually combines different disciplines, with each discipline developing one or more analysis models. MDDS depends on validated analysis algorithms to verify the robustness of the process.

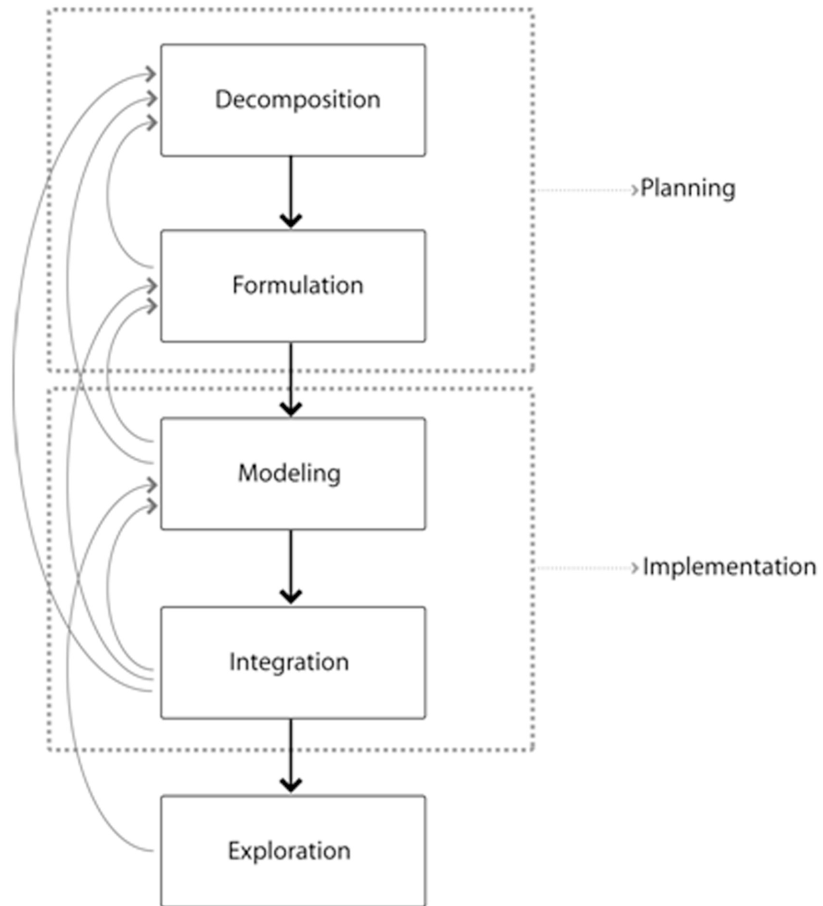
In order to enhance the level of automation in the process, optimization can be introduced. In case a specific design does not meet the original design requirements, it is modified and evaluated again in a search process for the best design possible, thus reformulating the design configuration.

Optimization models are design space search mechanisms. Searching the design space entails finding the best solution(s) within a domain of feasible solutions. An optimization model seeks to minimize or maximize an objective function by varying the values of design variables within an allowed domain. The choice of an appropriate search algorithm depends on several factors including the design synthesis model, the nature of the analysis models, the number of design variables, the existence of constraints, and the linearity of either the design variables or constraints.

In single objective optimization, the search direction can be well defined and a single solution, if it exists, could be found. However, in the real-world, decision-making problems are usually too complex and ill-defined, and have several possibly contradicting objectives. This implies that there is no single optimal solution but rather a whole set of possible solutions of equivalent quality (Abraham et al., 2005). Therefore, there is a need for evaluation models that can help in making decisions in the presence of trade-offs between conflicting objectives. As computational optimization processes evolve by generating and evaluating multiple design variations, they result in a group or 'point cloud' of optimized designs from the selection of good designs can take place based on design performance or other design viewpoints (Shea and Luebkeman, 2005).

Figure 1.3:

Proposed Framework for the Multidisciplinary Design System (MDDS)



Through a bottom up approach all the mathematical models that were developed as software components and modules are connected into a data flow network that includes clusters and subsystems. Software integration tools are used to satisfy the requirements of the MDDS process through efficiently automating the exchange of module information. The end result of a typical MDDS process is an integrated system model.

MDDS also supports the exploration techniques of the design space to help the lateral thinking among designers and to better describe and understand the complex relations between design variations and performance trade-offs. The MDDS can be continuously adjusted through several process iterations in order to investigate the influence of different parameter modifications. This aids designers by enhancing multidisciplinary negotiations which hopefully lead to better design quality.

While MDO techniques work on turning down the tradespace complexities into optimal solutions, the eventual goal of MDDS is to comprehend the tradespace itself, involving as much complexity as can be used by decision makers. This represents an alternate approach to MDO, which at the same time does not recognize qualitative design aspects.

The resulting MDDS is described by an evolutionary model moving from simple and generic ideas into further complex and detailed ones throughout the process. The system model is a dynamic and complex whole, interacting as a holistic structured functional unit. The system emergent properties are not detectable through the properties and behaviors of its modules, and can only be enucleated through a holistic approach. The solution found by this system is expected to be superior to the design found by solving and optimizing each discipline sequentially, since it can exploit the interactions between the disciplines.

The MDDS approach introduces a scenario where the idea that performance drives design is clearly identified. Performance based design, as a promising design paradigm, provides the basis by which design is guided through performance. The hope is that by incorporating the MDDS designers can gain a marketing edge through enhanced design quality and performance and improved collaboration among multidisciplinary design teams that would lead to reduced design time and cost.

In today's increasingly competitive market, design solutions that merely meet minimum project requirements are no longer guaranteed to prevail. Solutions must be cost-effective and generated through efficient multidisciplinary processes (Carty, 2002). An effective evaluation of these solutions involves therefore the integration of multiple disciplines. MDDS allows for identifying counter-intuitive solutions and functions of multiple design disciplines. Resultant integrated system models in MDDS are specifically tailored to problems in hand.

The concepts introduced in this thesis offer definitions that can be further enhanced. The computer products and experiments will act as system prototypes that can be used as starting points and developed further for more robust systems.

The prototypes that will be presented will focus on the early stages of design development. These include stages of conceptual and preliminary design. I will explore these stages as they are typically where most innovations and technological breakthroughs take place. At the same time, they are also where expensive, threatening

and hard to fix mistakes can occur. Major design solution approaches and design decisions are determined at the end of these stages. These decisions usually determine around 80% of the ultimate program cost and schedule (INCOSE 2002). This occurs due to the fact that the early design stages are where multidisciplinary trades matter the most. It is also where designers realize the significance of interdisciplinary relations and interactions, and where MDDS is highly expected to offer the most unambiguous short-term benefits.

Although the MDDS framework is domain-independent, prototypes will be developed for exploratory building design. I hope this will demonstrate the potential of such a computational design system and will provide a proof of concept for the framework presented in this thesis.

1.4 Thesis Structure

In his famous 1956 paper, the cognitive psychologist George A. Miller showed many coincidences between the channel capacity of human cognitive and perceptual tasks. The effective channel capacity is normally equivalent to a number between 5 and 9 equally-weighted error-less choices, which represents 2.80735 bits of information on average. Miller, not drawing any strong conclusions, hypothesized that the recurring sevens might represent something deep or just be a Pythagorean coincidence (Miller, 1956).

Following Miller's hypothesis, and given my own limited capacity, I have decided to loosely partition the thesis into eight chapters (not counting the introduction and conclusion), the contents of which intersect necessarily (figure 1.4). The chapter descriptions follow.

1. Theoretical Background. This will provide a literature review of design science theories of interest as well as a background on the emergence of system thinking and system theory and its influence on the design process.

2. Decomposition. This represents the first step of the MDDS framework and proposes approaches for partitioning design artifacts as well as design processes.

3. Formulation. This section will present some of the methods used for process modeling in different disciplines with an emphasis on methods used in both system and software engineering. The MDDS architecture formulation will be discussed later in the MDDS framework.

4. Modeling. Here the concept of mathematical modeling for design

will be presented. Each model's input, output, structure and algorithms will be introduced. These models will include synthesis, analysis, evaluation and optimization.

5. Integration. This chapter will address issues of integrating different distributed models and components for building the MDDS. Several software technologies will be discussed and presented.

6. Exploration. After the MDDS has been built we can start experimenting and exploring the design space. This chapter will present some methods that can be used to better understand the design space and the solutions generated.

7. The MDDS Framework. This chapter will present a framework for building the MDDS. All the five stages will be incorporated in this framework. The system behavior and evolution will also be discussed. In addition, implications of the MDDS on the design team and the computational design environment will be presented.

8. Experiments. This chapter will showcase the prototypes developed using the MDDS. It is intended to give a sample of successful applications of the described technologies.

The following chapters will attempt to address and answer some of the following questions:

What is design science?
What are systems and systems theory?
How can we decompose the design artifact and the design process?
How will we formulate the design system architecture?
How will we model the different design activities?
How will we integrate the different models into a coherent system?
How will we use the system to assist in exploring the design space?

Furthermore, the MDDS is expected to pose new challenges such as:

What is the expected behavior of the MDDS?
How will it evolve with the evolution of the design?
Will it require new design tools and environments?
How will it affect the structure of the design team?

Figure 1.4:

A simplified network that includes some of the topics that compose the MDDS discussed in this thesis.

2. Theoretical Background

2.1 Design Science

It was argued by Simon (1973) that a science of design could exist someday where design can be discussed in terms of well-established theories and practices. Simon claimed that design should move through multiple stages from its then current pre-science stage in order to become a mature science. He described that mature stage as acquiring a state of discipline where a consistent body of scientific research and practice exists and encompasses law, theory, application, and instrumentation (Kuhn, 1970).

Dixon (1987) argued that in engineering design particularly, both education and practice are led mostly by expert empiricism and intuition and specialized experience without sufficient scientific foundation. He stated that design in this case is very different from disciplines such as physics, chemistry, or biology where theories can be tested by controlled experiments. He argued that design was more complex than other fields because it involved not only people and organizations, but also the natural physical world and the in-progress design, which refers to the to-be-manufactured-sold-and-used physical artifact of a system. This complexity also lies in design being a process, where processes are not the typical subjects of theoretical formulations.

Hongo (1985) defines a design science or a scientific study of design activities as a collection of logically connected knowledge such as design methodology and design technique. A detailed definition implies that design theory is a system of methodical rules that identify the procedures possibly expected to conduct a planned route towards achieving a desired goal. He classifies types of rules according to methods of thinking, such as intuitive or discursive, and according to goals and applications, such as methods for solution search, evaluation, and calculation.

Coyne et al. (1990) provide another definition of science and design. As opposed to science, which formulates knowledge through deriving relationships between observed phenomena, design can be described as an action that starts with intentions and uses available

knowledge to reach a specific entity whose properties should meet those original intentions. The role of design is defined as that which utilizes that knowledge to transform a formless description into a specific description of form. This description, known as the design solution, is generated pragmatically according to the capacity of knowledge available to the designer. It is a compromise, rather than an ideal or correct solution, that meets to some extent the original intentions.

2.1.1 Design as an Object and a Process

All artifacts in the surrounding environment are the result of designing. Design can imply many meanings. The word “Design” can be both a noun and a verb (figure 2.1).

As a noun it can refer to an artifact or object that is a system usually defined by its geometric configuration, the materials used, and the task it performs (Papalambros and Wilde, 2000). According to Bahrami and Dagli (1994), design involves the development of plans or schemes of action. Their definition implies that design can be that developed plan or scheme, whether it is just embedded in the mind of the designer or externalized as a drawing or model.

As a verb, design can refer to the actual decision-making activities or processes involved in generating that artifact (Eggert, 2004). These processes determine an object’s form according to the required functions. Simon (1973) viewed design as a problem-solving process, a natural human activity that involves searching through a state space. The states in this space represent the design solution.

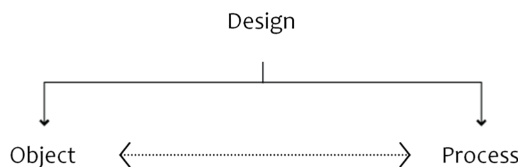


Figure 2.1:

Design can be considered both as an object and a process.

Coyne et al. (1990) define design as a purposeful activity that involves conscious efforts to reach a state of affairs in which specific characteristics are apparent. In this regard, design is initiated by recognizing the basic problem requirements. Being discontent with the existing state of affairs, the designer then becomes conscious that some sort of action should occur to correct the problem.

Louis Kahn, the famous architect, described design as a process where the inspirational forms of thinking and feeling generate form

realization (Bahrami and Dagli, 1994). To Kahn, thinking was considered a tool by which he would articulate feeling, his ideal mode of functioning, into expressive shape. He believed that the design process was understood intuitively by the creative mind as a single unified and consistent whole, as he used to synthesize elements from many sources into this whole rather than focusing on details of specific problems. He would pay more attention and dive into the core of the matter rather than going into finer-grain problems that were not really required at this point (Tyng, 1984). The question then becomes how a design evolves from fuzzy mental images and abstract generic concepts into a crisp design.

Design as a process is generally described as a systematic approach where the design process, as part of generating a product, is partitioned into general working levels. This allows for a transparent design approach that is both rational and independent of any particular field or industry. In this general approach, the problem is first analyzed, understood and decomposed into sub-problems. Sub-solutions are then generated and integrated to produce an overall solution (Cross, 1989).

There are several methods, intellectual frameworks, and tools that help support this process, including traditional engineering design (Pahl and Beitz, 1991), axiomatic design (Suh, 1990), and product design and development (Ulrich and Eppinger, 2000).

According to Papalambros and Wilde (2000) design is a complex human process that cannot be easily or completely described or understood. Therefore in the following chapters, will use models to help us define and understand the design process. Models are different from theories. According to Dixon (1987), a model does not establish a theory, but rather the theory is established when model behavior can be robustly explained through testing. Models then do not explain certain phenomena or predict specific behaviors, and so in a sense they are less ambitious than theories. They are however content with the provided explanations and predictions of phenomena, and can explain and sometimes replicate specific aspects of design behavior (Coyne et al., 1990).

In building useful models, the mathematical relationships between components are mostly required. It is easier for a designer to describe how a specific product is designed than to translate his behavior into a mathematical model unless a certain framework is developed for that purpose. The process of building computational models of design involves concepts from many disciplines such as artificial intelligence and problem solving, such as space search techniques, expert systems and neural networks, logic and fuzzy

logic, object-oriented methodology database, and language theory (Bahrami and Dagli, 1994). By implementing some of these concepts, our proposed design system can be better defined, studied and understood.

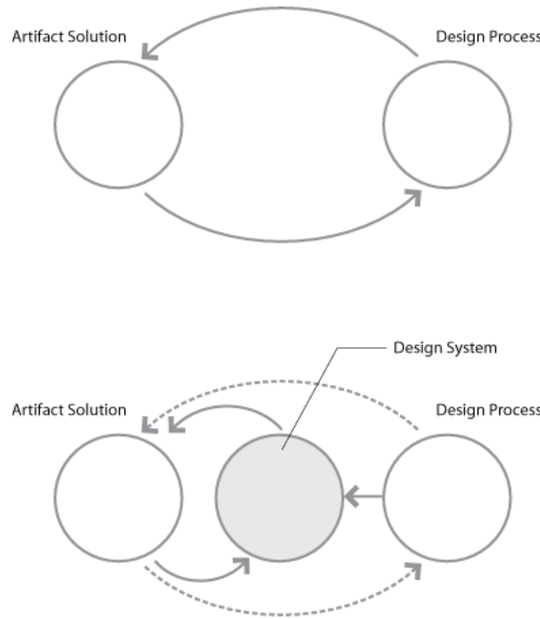


Figure 2.2:

The proposed design system should imitate the design process to produce the artifact.

As stated In the introduction, this thesis is concerned with creating a computational design system that attempts to imitate the design process to create the designed artifact (figure 2.2). Therefore, the primary focus will be on the notion of design as a process rather than design as artifact.

There is a relationship between the design artifact, which represents the *designed system*, and the design process, which is represented within the *design system*. We need to introduce a framework to map between both the design process and object. Since the system should lend itself well to computation, we need to build a computational model that that can assist in the design process.

2.1.2 Design Process Domains

How does design as a process map between the stakeholders' needs and the physical embodiment of those needs in an artifact? To understand this relation I will discuss briefly the Axiomatic design theory.

The early ideas of axiomatic design, developed by Suh and his colleagues at MIT, were first published in 1978 (Suh et al 1978), but the framework was initiated with the publication of the first

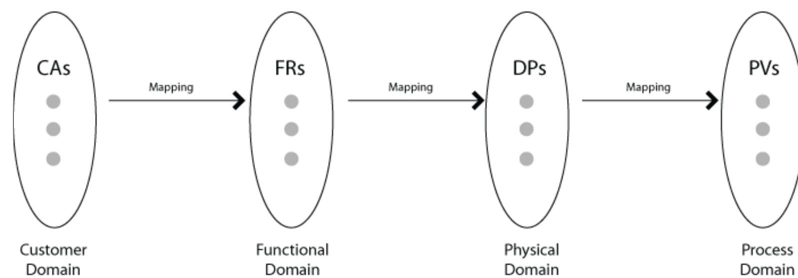
Axiomatic Design book by (Suh 1990).

The main motivation for establishing axiomatic design was primarily educational. It aimed at providing a scientific basis for the field of design in general (Suh 1990) to make the process of teaching and learning design more systematic and generalizable, based on the belief that designers should learn good decision making built on a scientific basis, and that there are core axioms that govern the design process.

Design has been described by Suh (1990) as the process of creating product solutions that satisfy customer attributes through mapping functional requirements in the functional domain into design parameters in the physical domain. According to Suh (1990), design thus is an interaction between what is to be achieved and how to achieve it in four domains: customer, functional, physical and process (figure 2.3). The success of the product in this sense especially in the marketplace is determined by the degree to which the functional requirements are met in the solution according to the decisions that the designer made. In the design process, functional requirements are considered negotiable aspects.

Figure 2.3:

Four design domains in the axiomatic design (Suh 1990)



The elements pertaining to each domain, mentioned earlier, are the customer attributes (CAs), functional requirements (FRs), design parameters (DPs), and process variables (PVs). The domain on the left relative to that on the right indicates what the problems are, or the objectives that are to be achieved. The domain on the right denotes the solutions, or ways to achieve those objectives. So in this case, CAs are to be satisfied by corresponding FRs that are the results of mapping CAs from the customer to the functional domain (Mullens et al., 2005). The general goal in any design problem is to choose DPs that determine FRs which consequently maximize satisfaction of the CAs that are subject to relevant design constraints (Cunningham, 1998).

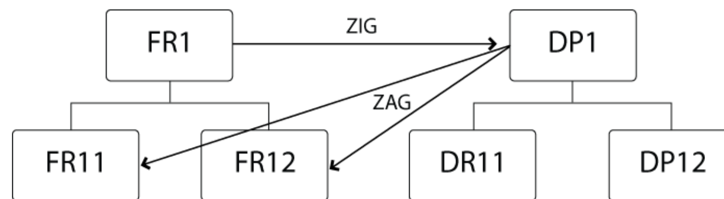
Several axioms were originally proposed (Suh et al. 1978), but later

some were viewed as redundant and were eventually integrated or completely removed. Suh (1990) identified two of these axioms as fundamental, through examining common elements in good designs related to products, processes, and systems. These axioms were believed to provide designers with a tool that structures their thought processes in early design stages.

The first axiom, known as the Independence Axiom, states that the independence of FRs must be maintained, implying that design decisions should be made without breaking the independence of each FR from the other requirements. The FRs therefore must be independent to each other and they should be reduced in number in order to be just sufficient to characterize the design (Suh 1990; Cunningham, 1998; Mullens et al., 2005). The strength of this axiom comes from the fact that it encourages designers to look for solutions that satisfy FRs independently, and so the issue of managing interactions is resolved.

Figure 2.4:

Zigzagging process between functional and physical domains.



The second design axiom, known as the Information Axiom, dictates minimizing the information content related to the task of fulfilling FRs in the design (Suh 1990; Cunningham, 1998; Mullens et al., 2005). The design chosen from the pool of alternatives satisfying the first axiom and at the same time comprising minimum information content is thus considered the *best* design. Information content is defined by axiomatic design as the log inverse of the probability of success to satisfy the FRs based on both axioms (Suh 1990).

Two governing rules in general should be maintained according to axiomatic design during this process in order to achieve good design. The first one implies following the two design axioms discussed earlier, while the second implies performing the zigzagging principle during decomposition (Mullens et al., 2005).

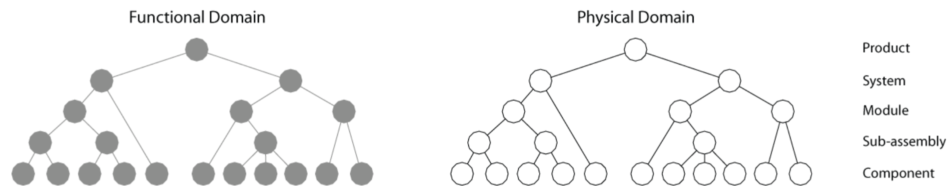
The zigzagging principle, illustrated in figure 2.4, guides the decomposition process from a high level to low detailed levels, thus guiding designers to zigzag between domains when they design. In the mapping between functional and physical domains, lower level FRs should be derived from the higher level FR while taking into

account the corresponding DP of the higher level FR. This means that designers should decide the corresponding higher level DPs that satisfy the FRs before the higher level FRs are decomposed into further sub-requirements (Cunningham, 1998).

In figure 2.5, mapping at any level takes place as the functions are assigned to discrete physical elements, shown by the arrows going from left to right, implying that the architecture is dictated. The physical solution is then utilized to guide decomposition in the functional domain. This is denoted by the arrows that go back to the functional domain horizontally and vertically only in the functional domain.

Figure 2.5:

Functional domain and physical domain hierarchies.



Design is thus a process of mapping between domains which develops a hierarchy from the system level to the detailed component level. The term mapping here refers to the process of translating customer needs into technical specifications, or functions. These functions are then translated into physical design characteristics or attributes, and then finally into process attributes which generate the requirements for production (Cunningham, 1998).

Designers usually map many functions to one high level element of the product, rather than decomposing functions and mapping them to the associated physical elements.

The way by which the functional hierarchy maps to the physical domain can be more complex than initially intended by designers. This creates supplementary interfaces between physical elements whose influence on function is almost impossible to predict.

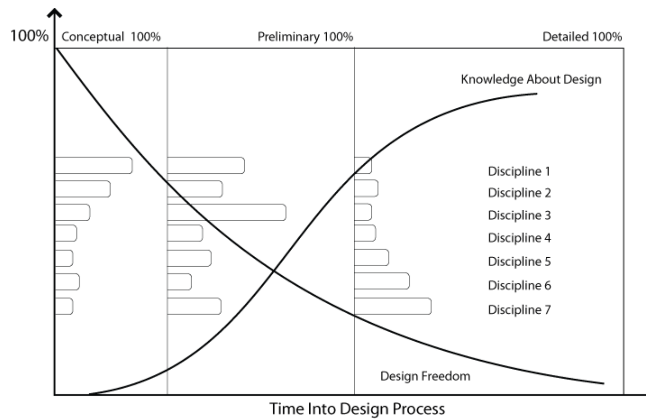
It should be noted however that the mappings between the customer and functional domains and physical and process domains are loosely structured and defined, as opposed to the mapping between functional and physical domains.

2.1.3 Design Process Evolution

The design processes is an evolutionary process which occurs between the time when a problem is assigned to the designer and the time the design is passed on to the manufacturer (Dasgupta, 1989). During this period the design evolves and changes its form (figure 2.8).

Figure 2.6:

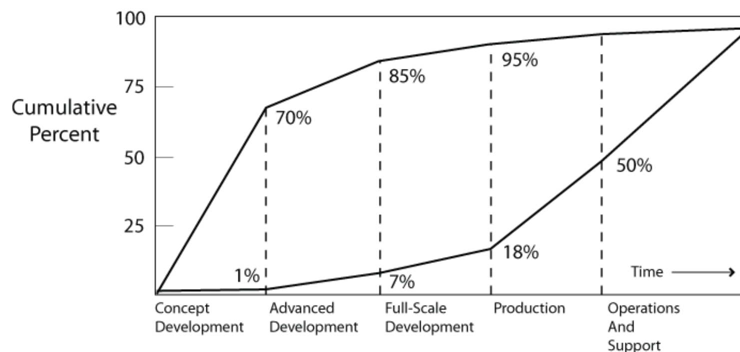
Short conception design phase with unequal distribution of improved quality and integrated disciplines for optimization.



However, throughout this progression of design evolution, there is an inherent relationship that is the core of all design development processes: the inverse relationship between design knowledge and freedom.

Figure 2.7:

Life cycle-cost committed versus incurred by life-cycle phase.



As the design evolves, design freedom rapidly decays while knowledge about the design object continuously increases. As the process moves forward, designers gain knowledge but lose freedom to act on that knowledge, as illustrated in figure 2.6. This has a key effect on the control of life cycle costs which are determined by the design concept and are very difficult to change significantly past this stage, as illustrated in figure 2.7.

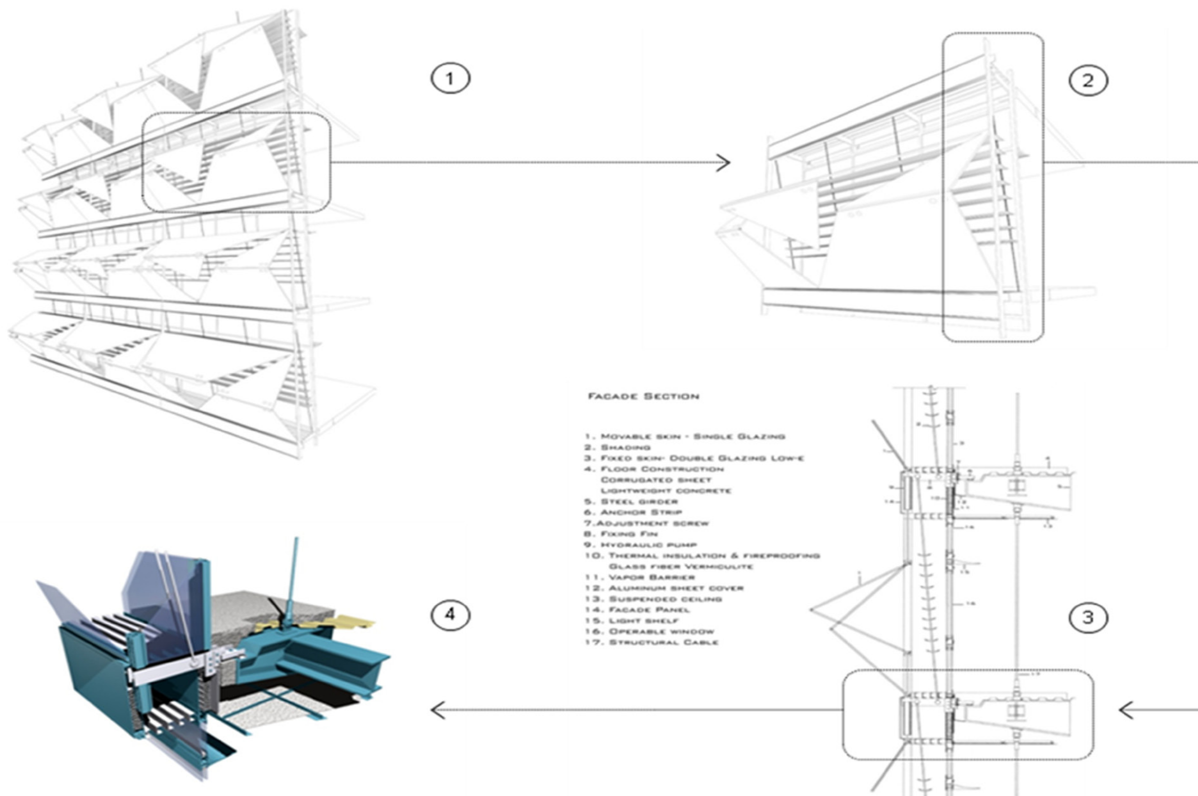


Figure 2.8:

An example of a building skin component. Knowledge about the design is increased as the design evolves over time.

Duvvuru *et al.* (1989) provided a classification of the design process comprised of four categories: *creative design*, *innovative design*, *redesign*, and *routine design*. In the creative design category, there is no *a priori* plan for the problem solution. In this case, design is considered as an abstract decomposition of the problem into levels that represent choices for the problem components. The main focus in this category is the transformation from the subconscious to the conscious.

In the innovative design category, the decomposition of the problem is known, but the alternatives for each of its subparts do not exist and must be synthesized. Design can be an authentic or unique combination of existing components. Duvvuru et al. argue that creativity plays a role to a certain extent in this category. In the redesign category, an existing design is altered in order to meet the required changes in the initial functional requirements.

In the last category of routine design, there is an *a priori* plan of the solution. The subparts and alternatives are known ahead as a result possibly of either a creative or innovative process. This type of

process deals with finding for each of the subparts the suitable alternatives that satisfy the given constraints. Duvvuru et al. (1989) show that the design process is more fuzzy, spontaneous and imaginative at the creative end of the spectrum, while it is more precise, crisp, predetermined, systematic, and mathematical at the other end which represents routine design (figure 2.9).

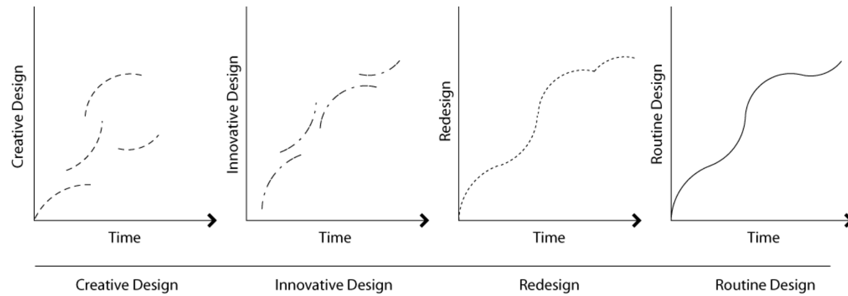


Figure 2.9:

At the creative end of the spectrum, design is very fuzzy. As it moves to routine design, it gets precise, crisp, and predetermined (Bahrami and Dagli, 1994).

2.1.4 Design Process Activities

Human problem solving including design is done using an iterative process (Simon, 1973; Asimow, 1962; Cross, 1989; Steadman, 1979). Designs typically evolve through a cycle that involves a *synthesis* activity and an *analysis* activity which is also known as the *generate and test* cycle (Rowe, 1987).

There is a fundamental difference, however, between synthesis and analysis design activities. To apply analysis, we are first provided with well-structured information about the object under study, and then we are asked to predict its behavior. This information is usually related to the form of an object such as shape, configuration, size, material composition, or even manufacturing processes. Through studying basic sciences and mathematics, object behavior can be modeled as a function of some input data. Predicted behavior in this context is generally the solution to an analysis problem (Eggert, 2004).

In the design synthesis activity, information pertaining to the desired function is provided, and the resulting solution to the design problem is concerned with form. In this case, there are many possible design solutions that can satisfy the desired function, and therefore design synthesis problems can have more than one solution, as they are more open-ended. There is no one structured procedure that can guarantee that unique solution. This happens due to the fundamental difference between analysis and design synthesis

processes. Information about the object of design is *ill-structured*.

Research in design methods suggests that designers first generate (or synthesize) a design proposal in response to the client's brief statement of requirements. After a proposal is generated, it is then checked or *analyzed*. If the design does not fulfill the requirements, a new design has to be synthesized (Cross, 1989). This happens through a loop of *refinement*, which can be very complicated and can turn out to be the most time-consuming part of the design process. This continuous *iteration* and evolutionary process can lead to a closed loop of decision-making, where refinements in one part of the design result in modifications or problems in other parts (Cross, 1989).

Minsky suggests the need for an additional mechanism which he terms the *progress principle* (Minsky, 1988). This is an *optimization* activity that guides the search and refinement rather than blindly generating all possible solutions. From this combined cycle, design could be considered an optimization process, as stated by Simon (Simon, 1973).

2.2 System Theory

2.2.1 System Concepts

Papalambros and Wilde (2000) define a *system* as a collection of entities that perform a specified set of tasks. For example, an automobile is a system that transports passengers. Schmidt and Taylor (1970) define a *system* as a collection of entities, such as people or machines, which act and interact together toward the accomplishment of some logical end.

Purposeful action is a key feature of any system. Sage and Armstrong (2000) define a system as a group of components that work together for a specified purpose. This implies that any system has to perform specific tasks that achieve its purposes. Systems are sometimes categorized in this context according to their ultimate purposes, which could be service-oriented (such as an airport), product-oriented (such as an automobile assembly plant), or process-oriented (such as an oil refinery).

A system is shown to be very perspective-dependent, where different components of the system could be grouped according to different perspectives to build up different notions of systems (Sage and Armstrong, 2000). In the engineering of the system, it is thus important to carefully define the nature of the system, its exact

scope of components as well as the interfaces to it.

Law and Kelton show that, in practice, the objectives of a particular study determine what is meant by a *system*. The collection of entities that constitute a system for a specific study may be only a subset of the overall system for another. Law and Kelton thus define what is called the *state* of the system, which is that group of variables that describes a system relative to the objectives of a study at a particular time (Law and Kelton, 1999).

A system in general has a group of basic characteristics. Any system should satisfy certain *functions* and consist of *objects* that are the physical or abstract parts, elements or variables within the system. It also consists of *attributes* which define the properties of the system and its objects, and *internal relationships* among its objects. In addition, any system exists in an *environment*, such that a system consists of a group of entities that affect one another within an environment and build up a larger pattern that is different from any of those initial entities.

In general a system includes the following features: wholeness and interdependence (the whole is more than the sum of all parts), correlations, causality, inputs/outputs, chain of influence, hierarchy, self-regulation and control, interchange with the environment, the need for balance/homeostasis, change and adaptability, and equifinality. (Littlejohn 1998).

A useful approach to understand systems is *system analysis*. System analysis was developed independently of systems theory. While systems theory models changes in a network of coupled variables, system analysis applies systems principles for the purpose of helping decision makers with several problems pertaining to systems. These include identifying, reformulating, controlling, and optimizing a system. Systems analysis takes into consideration diverse objectives, constraints, risks, costs, benefits and resources, and works to identify possible courses of action.

A system usually operates under causality, where the system tasks are performed due to some kind of stimulus or input (Papalambros and Wilde, 2000). This implies that these inputs have a significant effect on the system behavior. What actually constitutes an input or output relies primarily on the viewpoint from which the system is examined. Each systems viewpoint in general is based on a specific level of knowledge of the components of the system and its internal structure, the complexity of the system performance in relation to the environment, in addition to other engineering and management

issues. A system is analyzed at a specific level of complexity that corresponds to the interests of the individuals studying it.

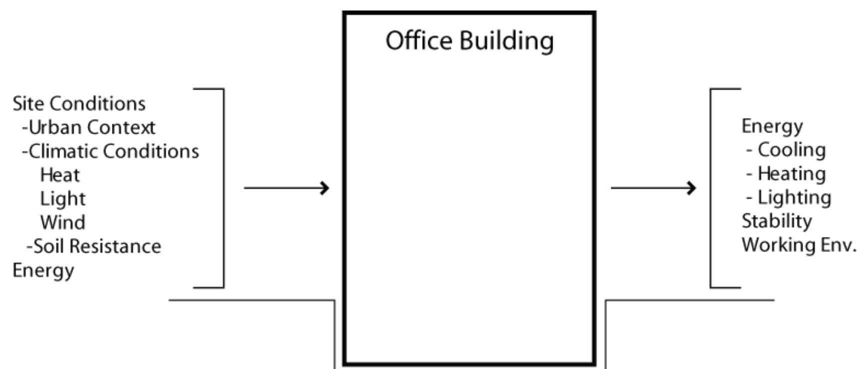
Usually designers perceive given problems subjectively in issues related to purpose formulation, and the conceptualization of linked system components that operate in constrained environments. It is thus significant to identify a system that is relevant and responsive to the problem being addressed. This can be achieved through several concepts, such as system-environment boundary, black box approach, component integration, and system state.

A *system boundary* around any subsystem is the entity that cuts across the links with the system environment and determines the input/output characterization. Figure 2.10 shows an example of a building system boundary.

Boundaries are crucial to defining architecture as they identify the deliverables and responsibilities of the different design teams, and at the same time they define what exactly is fixed or constrained at the boundaries. Anything that crosses that boundary must be facilitated by an interface.

Figure2.10:

The boundary around an office building system determines its relation with the environment



The black box behavioral approach, represented schematically in figure 2.11, is used when little or nothing is known about the composition of a specific system or its internal connections. The system is analyzed in terms of the black box system response to any given input, where the system device is considered as an all-embracing impenetrable black box (Meredith et al., 1985).

In many cases, it may be more suitable and convenient to consider a system as a black box system although there may be partial knowledge about it. This is due to the fact that in many design situations, the relevant and required information is more related to

system performance as a whole in terms of the relationships between system inputs and outputs, rather than the complex interrelationships among internal system components and their own individual behavior.

Figure2.11:

The black box approach identifies system performance in terms of inputs and outputs.

Once a clear and explicit relationship exists between a group of elements belonging to known input variables and another belonging to required solution output variables, a systems problem can be solved. The behavior of the black box system is depicted and analyzed in terms of the changing values over time of both groups. There is no need then for the knowledge of the internal structure of the system components once the functional performance criteria for the input-output behavior have been established.

In most design and engineering problems, the required solution is achieved through the construction of a group of physically linked components (that can be considered black boxes) that are connected in a specific configuration. The system components in this case consist of specifically designed and built-in functions and have known attributes. The configuration that integrates the components together, which denotes the component integration approach, determines the intrinsic structure and behavior potential of the whole system (Meredith et al., 1985). An important feature of this approach is how interrelated problem and solution components interact with each other. Aspects of a given problem are sometimes a function of how these components interact (Sage and Armstrong, 2000). This approach is also applicable to design situations that deal with work processes and planned sequential actions, and not only physically connected components.

In the component integration approach, the whole is not simply an aggregation of individual components. From a systems point of view, it is important to distinguish between different components or subsystems due to the complexity of the system. This is usually done by organizing the different components into groups based on function or another principle, such as organizing systems into hierarchies.

Given that the behavior of a complex system changes frequently

over time, systems engineers usually use the concept of the state of a system for analysis and modeling purposes. State is a collection of variables that can be considered a snapshot of a particular system at an instant of time (Sage and Armstrong, 2000). The main interest of the state theory approach is the description of the state of the system and the detection of changes in the system state according to new inputs. The effort lies in describing the internal responses of the system in terms of a minimal consistent set of system indicators.

The *state vector* provides a simple approach to understanding system behavior, as it denotes a set of reference variables that give a descriptive measure of the system state at any given instant in time (Meredith et al., 1985). Each reference variable is known as the *state variable* of the system. The systems view of the state of an airport, for example, can include variables such as the number of planes waiting to land or takeoff, or the number of available parking spaces. The system behavior then depends primarily on the changing values of each of these variables, whether they are captured continuously or at discrete times along the life of the system. The nature of a state variable and the number of entries in the required state vector are closely coupled with the purpose being modeled and the complexity of the system being formulated. Choosing a set in particular out of many variables enables the development of a system description. Choosing a different set may allow a totally different system description. For example, if the concern in the state of the airport is enhancing passenger convenience, this would probably require modeling parking spaces. If the concern however is enhancing air traffic safety, car parking spaces would not be relevant, and therefore would not be modeled (Sage and Armstrong, 2000).

2.2.2 System Architecture

Every system has an architecture, which in essence strongly affects its behavior (Crawley et al., 2004). Architecture is significant in a variety of disciplines and in many technical fields. The typical connotation describes civil architecture of buildings, but the term also extends to include physical products, engineering systems, and infrastructures, in addition to informational artifacts such as software and computer networks.

"Architecture" in Webster's Online Dictionary is a "formation or construction resulting from or as if from a conscious act," or "a unifying or coherent form or structure". Crawley (2003) describes a generic architecture as "the conceptualization, description, and design of a system, its components, their interfaces and relationships

with internal and external entities, as they evolve over time".

There are multiple definitions for "architecture" which are largely dependent on the context in hand (Hastings, 2004). Different disciplines look at architecture from different perspectives. Such disciplines include product development, mechanical systems, engineering systems and others. From a product development viewpoint, for example, architecture is described by Ulrich and Eppinger (2000) to be an "arrangement of the functional elements into physical blocks". In the engineering systems field, system architecture is defined by the ESD Architecture Committee at MIT as "an abstract description of the entities of a system and the relationships between those entities" (Crawley et al., 2004). They also state that architecture, which embraces meanings such as an "arrangement of entities and relationships between them" or as "relationship between form and function", represents the physical embodiment that the designer finds in order to perform the required functions of the design problem.

These definitions share many things in common. The basic common characteristics of architecture include the description of the system elements, the functional character of these elements, and the structure of the interrelationships among them. Every discipline, however, differs in terms of the specifics of what those parts are and how accurately they are connected together. Another characteristic involves the development of system concepts, which comprise internal form and function, while taking into consideration at the same time the holistic view and thinking out of the box (Crawley, 2003).

Defining an architecture for a system serves many goals, such as abstraction, reducing the impact of continuous changes, and facilitating communication (Zachman, 1987). An architecture abstracts complex systems through describing simple models. This abstraction enables the definition and control of interfaces and the integration of system components. An architecture also enables reducing the impact of changes to fewer steps especially in redesign processes. It focuses on parts that require major change. As an architecture offers multiple abstract views on the system, it provides a means of communication during the design or re-design process, where useful discussion occurs to represent the perception of each communicating party of the problem in hand.

Perry and Wolf (1992) draw an analogy between system architecture and the architecture of buildings. They describe how architecture provides multiple views, abstractions, architectural styles, and how

engineering principles and materials significantly affect the architecture of a building. A building architect's interaction with a client versus a contractor for example, the architect provides different views of the building in which there is a focus on some specific aspect. He provides elevations and floor plans in addition to scale models for the client in order to give him a good impression of the building. The contractor however is provided with the same floor plans in addition to structural views that provide detailed information about diverse design considerations.

Architectures can arise within a variety of mechanisms (Crawley et al., 2004). These include the deliberate design of a system from scratch, the evolution of a design from previous designs with strong legacy constraints, obeying regulations, standards, and protocols, the expansion of smaller systems with their own architectures, or the exploration of form and behavior requirements through dialogue between architects and users.

2.2.2.1 Form and Function

Form

The determination of form to satisfy and execute a required function represents the essence of design. Eggert (2004) defines form as what the product looks like, what materials it is made of, and how it is made. He identifies the basic characteristics of the form of a product to be shape, size, configuration, material, and the manufacturing processes used to make the product.

Form, according to Crawley (2003), refers to the physical or informational embodiment that exists or has the potential to exist. Form represents the thing that is eventually implemented and operated in a solution specific domain. Implementation here can include manufacturing, building, writing, composing, etc. Operation can refer to running, repairing, updating, etc.

Form can be represented as the sum of elements and structure, where elements are segments of the whole of the form, and structure denotes the formal relationships among the elements.

Function

Form is intimately related to function. The quotation by the famous architect Louis Sullivan, "Form ever follows function", supports this idea that the form of an object is highly dependent upon the function

it performs. Similarly function is associated with form and emerges as form is assembled, as well as when different sub-functions are aggregated together yielding what the whole system eventually "does" (Crawley, 2003).

According to Eggert (2004), the function of a product is what it is expected to perform. Crawley (2003) defines function as a product or system attribute, conceived by the architect, that denotes the activities, operations and transformations that cause, create or contribute to performance and meet the required goals. Ideally, function is expressed in a language that is solution neutral.

There are three fundamental entities, or *functional building blocks*, that compose the media on which systems operate and function (Kossiakoff and Sweet, 2002). These are information, material and energy. Information refers to knowledge content and communication. Material refers to the substance of physical objects, while energy boosts the operation of the active system components. The physical embodiment of individual functional elements is thus usually configured through the construction of material, the control of external information, and the power of a source of energy, regardless of the primary function and classification. Information can be further subdivided into two classes. The first class involves *signal elements* that sense and communicate information, such as radio signals. The second class involves *data elements* that interpret, analyze, organize and manipulate information, such as computer programs. This results in a total of four functional blocks. System functions usually perform a purposeful alteration in some of the characteristics of these building blocks. Therefore these blocks are considered fundamental for identifying and categorizing the main system functional units.

As our main concern is man-made architectures in complex systems, it is important to understand that these systems have specific *primary functions*, in addition to other properties known as *ilities* (Crawley et al., 2004) which include adaptability, durability, maintainability, flexibility, etc. Primary functions denote the immediate value of a product or system, such as flying for airplanes, delivering products for companies, and so on. Ilities have life-cycle value that describes properties of "performing things well".

Designing complex systems that accomplish all primary functions and all ilities is inherently a complex task. There are often compromises that have to be made when it comes to conflicts between desirable short-term properties and life-cycle properties. The system architecture strongly affects how ilities are achieved,

how they internally interact with each other, and how they interact with primary functions. These systems will require additional resources, leading most probably to increased system complexity, which can cast doubt on the benefits of these typical architectural decisions. Their results may only be fully known in the future. De Neufville et al. (2004) discuss some methods for assessing such uncertainties and relevant precautions.

Another significant factor that can increase complexity is that architectures may evolve over time, especially in prolonged systems such as infrastructures. Systems vary in their response over time. Some systems serve their intended function successfully throughout the whole long life cycle. Other systems perform outstandingly with time concerning the original function and handle more functions that were not perceived in the initial design. Others do not fulfill the original function and quickly run out of service, thus being unusable for other functions.

System architects should develop systems that can adapt and grow within the initial rules and structural arrangements in order to reduce the unfavorable severe constraints that are inherited from the original conditions.

2.2.2.2 Architectures of Integrality and Modularity

Mapping function to physical elements (Form) within hierarchical structures is significant in design theory as discussed earlier. Ulrich and Eppinger (2000) define two categories of functional mapping related to product architecture, which refers to the scheme by which functions are mapped to physical elements and the internal interactions between those elements are defined. These categories are modular architectures and integral architectures.

The basic characteristic of modular architectures is the relatively strong and direct one-to-one mapping of functional elements to physical elements. Since the role that interfaces play for each function among the different physical elements is well defined, modular products become more appealing. Moreover, individual physical components can be designed relatively independently by functionally decoupling them. Downstream integration throughout the design process thus becomes less complex.

Integral architecture, however, involves a complex mapping of functions to physical elements. There is no direct mapping and the interfaces of physical elements acquire complex relations to functions (Ulrich,1995). The consequent effects of element

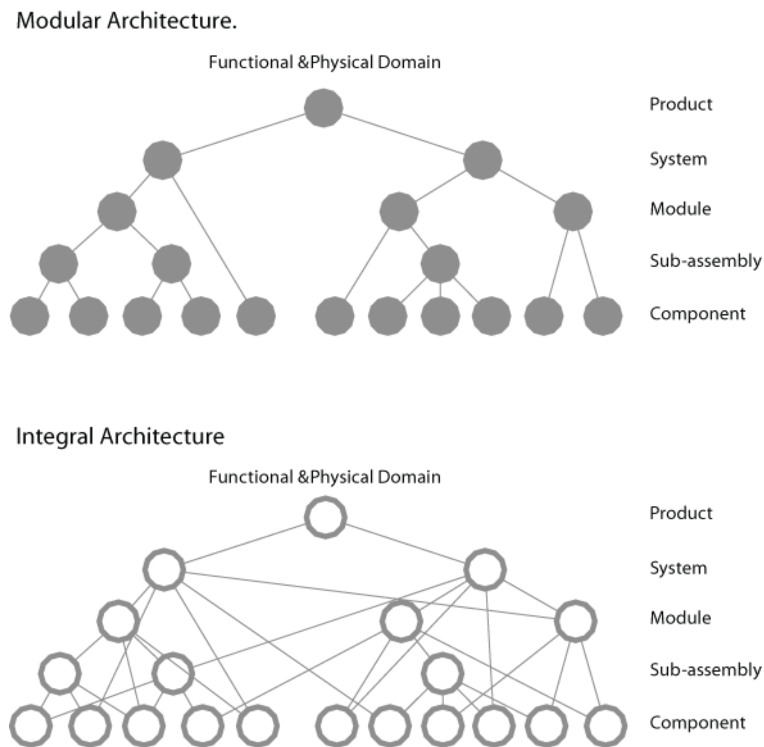
interactions on functions are hard to recognize, or *incidental* (Ulrich and Eppinger, 2000). Functions in *inherently integral* products are delivered in a coupled fashion, meaning that modifications in a part, feature, or sub-element of a product affect the global system performance in many functions. The term "inherently integral" is used here to refer to products that have many functions shared by many of the same physical elements.

As illustrated in figure 2.12, the lateral lines that run among physical elements denote the distributed nature of functions among a variety of elements. This is not the case in modular design where there is a close match between the functional and physical hierarchies.

Figure 2.12:

In modular architecture there is a close match between the functional and physical hierarchies.

In Integral architecture functions are distributed among a variety of elements.



Some design theory literature considers modular architecture ideal and considers a design to be inferior if designers could not achieve modular design. However, what occurs in reality implies that designs with integral characteristics can represent a higher degree of success and goal accomplishment by their designers (Ulrich and Seering, 1990; Whitney, 1996) (figure 2.14). In a real design problem, designers are faced with many goals to achieve. These goals often conflict with each other and cannot all be attained equally well (figure 2.13). Resolving those conflicts to a reasonable degree should be acknowledged rather than blaming designers for failing to achieve a

modular design.

It is important to consider the relevance and use of both modular and integral architectures when it comes to the field of research in product development. Integral architectures can be deployed in the case of simple products (Ulrich and Ellison, 1999) and even in complex inherently integral products which do not conform to ideal models or where modularity is not desirable. Even when considering engineering issues, integral architectures are still relevant. Modular architectures are needed, however, and become more relevant in situations where strategic issues are included, such as outsourcing and new architecture development.

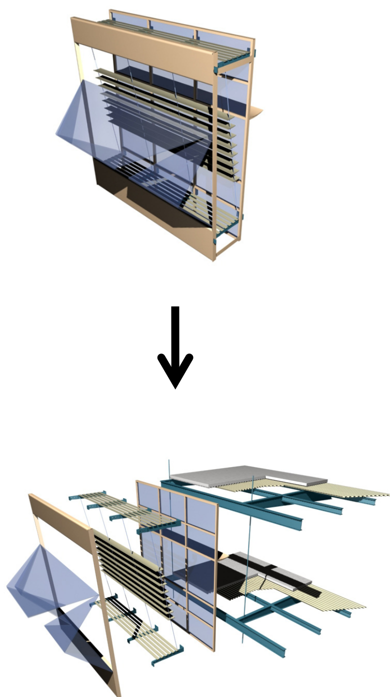
Therefore, in short, the modular scheme can be viewed as one which exhibits strategic goals such as additions, adaptation, flexible processes, and diversity of production (Ulrich, 1995), due to direct mapping. The integral scheme, however, accommodates better overall performance at the expense of strategies (Ulrich and Seering, 1990 ; Whitney, 1996).

Figure 2.13:

An Example of two Building Skins with one representing a modular architecture and the other representing an Integrated architecture.

Project Credit of Integrated Architecture Skin: Anas Alfaris, Alexandros Tsamis.

Modular Architecture



Integrated Architecture

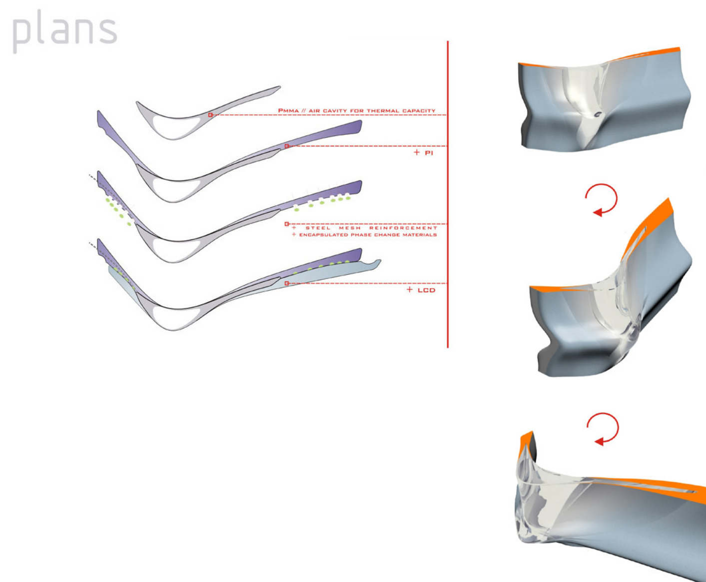


Figure 2.14:

Modular Architecture in physical product design is not always superior to integrated architecture as is illustrated in this nail clipper example (Ulrich, 1995).

2.2.2.3 System Structure

Structure describes the relationships among system objects (Crawley, 2003). It can describe connections that take place both in form and in function while operating. Connections that are descriptions of form include concepts of spatial location, proximity, topology, or assembly process. Connections that are descriptions of function include flow of information, energy, and material. Products and systems are separated from other supporting systems and operands by a boundary.

In the next two sections, two main types of structural models will be introduced. First a discussion of hierarchies will be presented followed by a discussion of networks. Hierarchies represent partially ordered sets with more constrained relations while networks represent a more general and encompassing term, as they denote sets of entities with interconnections.

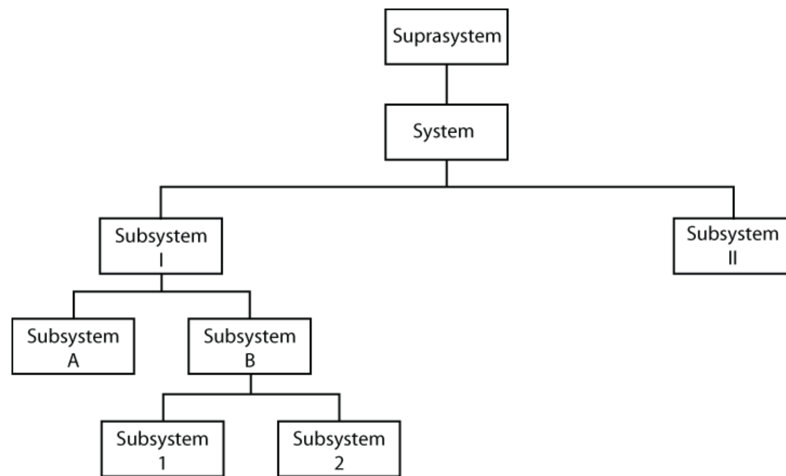
Hierarchies

Hierarchy theory is an emerging part of the work of researchers in different disciplines, such as the economist Herbert Simon, chemist, Ilya Prigogine, and psychologist, Jean Piaget (Allen, 1998). It descends from general systems theory, and it focuses on organization levels and scale issues. Hierarchy theory builds on simple principles to organize the behavior and structure of complex systems with multiple levels, as illustrated in figure 2.15.

Simon (1973) provides various examples of hierarchy. He mentions organizations, but at the same time points out that hierarchy does not necessarily imply top-down relations of authority. Simon basically sees that problems can be solved more easily when they are decomposed into sub-problems whose solutions can be combined into a solution to the problem as a whole.

Figure 2.15:

The structure of complex systems can have multiple hierarchical levels.



Kossiakoff and Sweet (2002) suggest that a complex system builds on hierarchical structures consisting of interacting sub-systems, which are further composed of simpler functional entities, and so on down to primitive elements, referred to as parts or components. According to Kossiakoff and Sweet, any system can be, in practice, applicable to various levels of aggregation of complex interacting elements. Every system can therefore be a sub-system that belongs to some kind of higher-level system, and every sub-system can be regarded by itself and its components as a system.

Simon viewed hierarchy as a general principle of not just complex structures, but of complexity in general, where he considers it the main form of architecture of complex systems (Simon, 1973). He argued that hierarchy emerges almost inevitably through a wide variety of evolutionary processes simply because hierarchical structures are stable (Agre, 2003).

Simon, while reflecting on general systems theory, points out that inferring the characteristics of one whole is a complex process given the properties of the components of that whole and their interaction laws to begin with. He concludes that "*an in-principle reductionist may be at the same time a pragmatic holist*", thus declaring the fact that the whole is pragmatically more than just the sum of its parts.

In this view, sub-systems may be regarded as quite complex on their own. They perform similarly to and acquire properties of a system. The fundamental difference is that the capability of performing a meaningful function requires the presence and functionality of other companion sub-systems for the general system to work.

Simon (1973) describes complex problems in terms of hierarchical

structures that consist of "nearly decomposable systems". In this concept of *near-decomposability*, the upper levels in hierarchy emerge due to the fact that their corresponding parts are not completely separate. The basic structure of this near-decomposable nature implies that the strongest interactions occur within groups while weaker (but not negligible) interactions occur among groups. In these interactions, the short-run behavior of each sub-system is almost independent from that of other sub-systems. The behavior of any sub-system in the long run, however, depends on the behavior of others only in the aggregate sense and not as individual components. Simon (1973) describes a variety of systems ranging from business organizations to biological systems that exhibit the property of being "nearly decomposable".

One of the basic features of nearly decomposable systems is that what connects any element at a hierarchic level with the relevant elements at the next lower level is actually the relation of a system (as a whole) and its elements (components or parts). Therefore, the systematic effect should take place from a level to its next higher level. This indicates that elements at different levels have different characteristics. When a level is traversed, qualitative change must occur.

Two main factors control how a system is perceived in view of hierarchical structure: constraints and possibilities. Constraints come from upper levels, while limits of possibility come from lower levels. To perceive a system hierarchically, one must pay attention to what is allowed by upper level constraints as a response to higher system purposes. This is due to the fact that the lowest level entities become constrained, losing degrees of freedom, and are held against the upper level constraint to give constant behavior. At the same time, the mechanisms that define the limits of physical possibility for the parts of the system to work as a whole have to be considered. Unless there is a distinction between these two factors, the concept of hierarchy becomes confused (Allen, 1998).

Hierarchies can generally be classified into nested and non-nested hierarchies. Nested hierarchies typically involve upper levels that consist of and are made of lower levels, e.g. an army is a nested hierarchy which consists of a number of soldiers who make up that army. On the other hand, the containment requirement is not that strict in non-nested hierarchies, e.g. a military command is a non-nested hierarchy with regard to army soldiers, as a general does not consist of his soldiers (Allen, 1998).

Mathematically speaking, hierarchy is a partially ordered set (Allen,

1998). In simple terms, hierarchy implies a group of parts inside a whole, containing upper levels that are above lower levels, and sustaining a relationship which is asymmetric between both levels. These hierarchical levels are occupied by entities which distinguish the identity of each level. An entity can reside on any number of levels, depending on the relationship between the hierarchical levels.

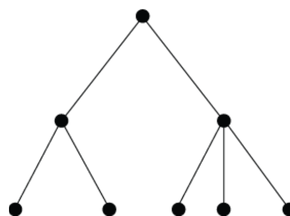
Hierarchies are typically associated with an ordering, that is a ≤ relationship (Magee et al., 2006). This partial ordering can be represented through depth of layer or numbered levels for each single node in the hierarchical structure. A ≤ ordering contains internal cycles. Nodes can have direct edges to their "brother" or "parent" nodes. Strict orders (a < relationship) however have no internal cycles.

Based on the topology and the relation between entities and levels, hierarchies can be classified into three basic types: tree structured hierarchies, which are sometimes known simply as hierarchies; layered hierarchical structures; and mixed or hybrid trees and layers (Magee et al., 2006).

Trees can represent small, medium and large structures in both human organizations and engineering systems. Tree structures are associated with top-down design. They are hierarchies that represent a reductionistic approach to decomposing problems into smaller sub-problems. Poor decompositions are likely though due to the generality of this approach. Tree structures are characterized by having exactly one parent in the immediate preceding level. The only exception is the root node.

Figure 2.16:

A tree with 8 nodes and 7 edges or links, 5 paths from root node to bottom or leaf nodes, 3 levels (Magee et al, 2006).

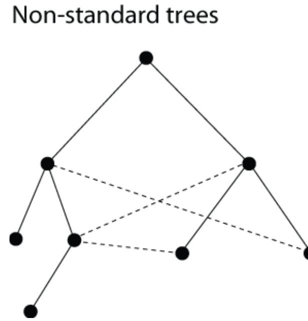


Pure trees are not considered relatively flexible. It becomes difficult to make internal changes or get around a non-functional node or edge while maintaining the same pure structure. An important aspect related to tree structures is that they lend themselves easily to competitive environments (Magee et al., 2006). Specific

subsystems may be assigned to individuals whose performance is judged in relation to others residing at the same hierarchical level. Figure 2.16 illustrates an example of a tree structure.

Figure 2.17:

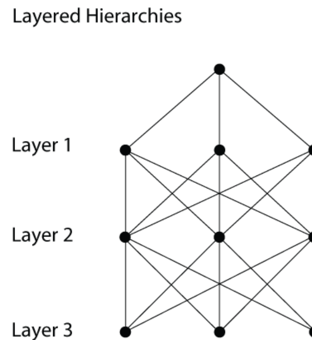
Non-standard trees: an impure relatively complex tree with non-standard interconnections (Magee et al, 2006).



Layered structures usually have multiple parents rather than just one parent in the immediate preceding level. They can also change parents readily. Layered systems therefore can attain high complexity, as there are many potential interconnections both between and within layers. For example, a layered structure with multiple layers and no horizontal interconnections can connect to all nodes in the layer immediately above or below it (Magee et al., 2006).

Figure 2.18:

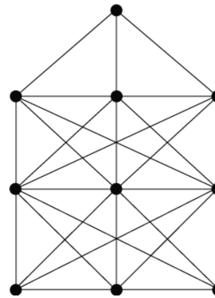
Layered hierarchies with horizontal interconnections (Magee et al, 2006).



Horizontal interconnections are typical of layered structures and not tree structures. They usually support teamwork and cooperation. Such interconnections add to the complexity of the nodes and consequently the overall system. Layer skipping is not allowed in pure layered structures. This does not usually constitute a problem, as the system is context aware. Hierarchies do not undergo cycles except within single layers. This is considered a modeling limitation, however, one cycle could be introduced to allow for feedback (Magee et al., 2006). Figures 2.18 and 2.19 illustrate examples of layered structures. Mixed or hybrid tree and layered structures are used in human organizations in addition to some technical systems.

Figure 2.19:

Three layers, a root node, 10 nodes, with horizontal interconnections (Magee et al, 2006).



Networks

A network represents a set of items with connections between them. The items are known in network terminology as vertices or nodes, while connections are known as links. Items can be assigned names, sizes and levels of significance, while connections can be assigned lengths and capacities.

The basic intrinsic characteristic of networks is their ability to represent systems, where the systems consist of items and their inter-relationships and connections (Magee et al., 2006). Items can be physical, such as locations or individuals, or abstract such as processes and tasks. Similarly, connections can be physical, like roads between different locations, or abstract, such as information flows between processes and tasks. Looking at networks as a whole, network representations themselves can be specific, in that they identify the different items and the different connections, or abstract, where they convey very little about the items or connections.

There are numerous types of systems that take the form of networks. Examples of such systems include the Internet, social networks between individuals, organizational networks, transportation networks, and many other types of networks.

In mathematical literature networks are mostly known as "graphs" and are considered one of the basic concepts in discrete mathematics. Euler's graph in the 18th century is probably the first true proof in graph theory which later developed into a substantial body of knowledge in the twentieth century (Newman, 2003).

According to graph theory, a graph is a pair of sets *V* and *E*, where each element of the set *E* is a two-member set whose members are

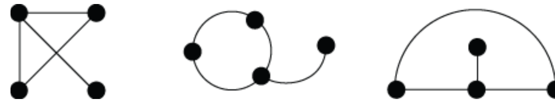
elements of V. The set V consists of vertices known as nodes. The set E consists of edges known as arcs, links or bonds. Edges are drawn as lines connecting two vertices at their endpoints. Graphs are mostly preferred to be perceived graphically. The following is an example of a graph:

$$V = \{a, b, c\}, E = \{\{a, b\}, \{a, c\}\}$$

The significant issue however in graph theory and networks is the pattern of connections and not the geometry (Hayes, 2000). These patterns include *connectivity, resource exchange*, and *locality* of action (National Research Council, 2006). Networks have well-defined *connectivity*, or connection topology, where each node has a finite number of defined dynamic connections to other nodes (figure 2.20). These connections between nodes exist only if there are one or more classes of resources, which are important and meaningful to the networked system that can be exchanged among them. This exchange of resources only occurs and is effective in local interactions, represented in node-to-node interactions.

Figure 2.20:

Networks with the same topology

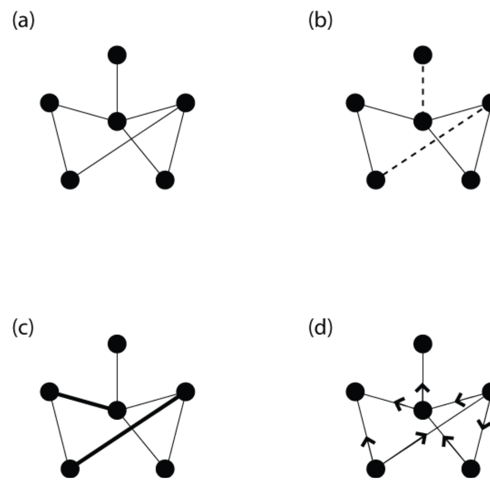


Network theory and graph theory have been used extensively to measure networks, discover connections, and determine how the flow of information, energy, and material between entities occurs (Sterman 2000). The current literature assumes often both identical nodes and links. Many authors (Watts and Strogatz 1998; Strogatz 2001) have studied graph properties to predict the resulting behavior if certain nodes are removed, or if control of the network's paths is decentralized.

There are many types of networks, some more complex than just a set of vertices simply connected by edges (figure 2.21). Edges point in only one direction, and are thus called directed edges. Graphs that consist of directed edges are known as directed graphs or *digraphs* (Newman, 2003), such as graphs representing email messages. Digraphs can be either cyclic or acyclic, that is they can contain closed loops of edges or not. Edges that connect more than two vertices together are known as *hyperedges*. Graphs that consist of such edges are called *hypergraphs*. Graphs that contain multiple edges connecting the same pair of vertices are known as *multigraphs* (Hayes, 2000).

Figure 2.21:

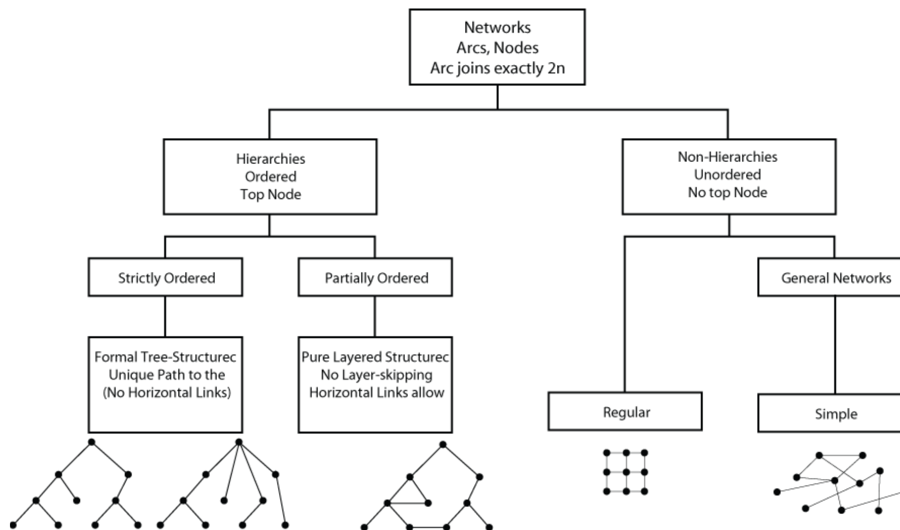
There are many types of networks which depend on the type of edges connecting vertices (Hayes, 2000).



Graphs can be metric graphs, where links have real lengths and node positions obey triangle inequality, or non-metric, which consists of just a logical layout. Edges can carry weights to denote strength or weakness of a specific relation. All vertices need not be connected by edges. Disconnected vertices or components can still constitute elements in a single graph. There may be more than one type of vertex or more than one type of edge in a given network. Vertices and edges may represent different types of associated attributes.

Figure 2.22:

Taxonomy of networks (Magee et al, 2006).



Graphs can also have other properties and types, such as changing over time, where vertices or edges appear or disappear occasionally, or their values are modified. There is still much to explore about the different possibilities and types of networks. Figure 2.22 shows different types of networks.

2.2.2.4 Behavior

The term "behavior", according to Eggert (2004) describes how a product actually performs. The aim of system design in general, and of architectural design in particular, is to achieve the desired behaviors that are outputs of functions plus ilities while predicting and limiting undesired behaviors.

Large complex systems have behaviors that are usually not attributed to their individual sub-components. Some of the behaviors are considered to be deliberate and intentionally developed through methodical design activity. These behaviors can be desirable or undesirable.

Other behaviors are mostly unanticipated and therefore known as emergent behaviors (Crawley et al., 2004). Emergent behaviors are very similar to what Ulrich and Eppinger (2000) identified as "incidental interactions". They exist when the system or its interactions with the surrounding context are not fully comprehended. They can exist due to other unpredictable factors, such as future system changes or the difficulty of modeling every single system state.

Emergent properties can be desirable or undesirable when thought of in retrospect. For example, automobiles were intentionally designed for personal transport purposes, but in retrospect, there are many behaviors that emerged later on. Unexpected behaviors included suburban growth, expansion in shopping malls, and developing a sense of personal freedom.

2.2.2.5 Process of Creating Architecture

Creating architectures is an important process in terms of generating working systems that fulfill desired requirements in a defined fashion within certain constraints (Crawley et al., 2004). Architecture is thus necessary to design and understand the behavior of systems. Architecture as an "arrangement of entities and interrelationships among them" determines a variety of ways in which the system behaves. Thus designers can use architecture to create systems with the desired behaviors. They can then structure them to facilitate the process of design and manufacturing. These processes can be conflicting however in some circumstances.

Although architecture is a necessary aspect in complex engineering systems, it is still not fully comprehended. There are no algorithmic procedures for generating architectures to serve desired behaviors,

in terms of selecting elements and linking them together. There are also no tools that could identify unintended emergent behaviors through looking at the behaviors of individual elements.

A powerful notion of process integration seems to be embedded in the definition of architecture, which implies all relationships between all system elements. The main focus however lies in determining what constitutes a system in terms of components, parts, and assemblies, in addition to how these components function and how they interfere with each other.

Systems have numerous architectures as well as different architectural hierarchies. This can happen because of the way the system boundary is defined. It can also happen because the system encloses a physical architecture in addition to various virtual architectures that catch significant views of system behavior. Most of these virtual architectures are consistent with mental models of different behaviors. Many representations are required to describe systems and their architectures.

As a system, the design system proposed in this thesis will have an architecture and will be made up of different parts that perform different functions. Within the proposed design system, the system can be viewed as being composed of small design cycles. Complex processes can be made up of many processes that are themselves made up of many other processes. The architecture of the system should relate these processes. It also should be comprised of modules that have interconnections between them that evolve over time.

The following chapters will attempt to address and answer the following questions: How will we decompose the design artifact and the design process? How will we formulate the design system architecture? How will we model the different design activities? How will we integrate the different models into a coherent system? How will we use the system to assist in exploring the design space?

3. Decomposition

3.1. What is Decomposition?

I use the term "decomposition" to refer to the act of breaking a large problem into a set of smaller problems or elements, whether it be a function that must be performed, a physical entity that must be designed, a design development stage, or a design cycle .

Smith and Brown (1993) point out that decomposition, where a problem is divided into simpler sub-problems, is the prototypical means of addressing complexity in design problems.

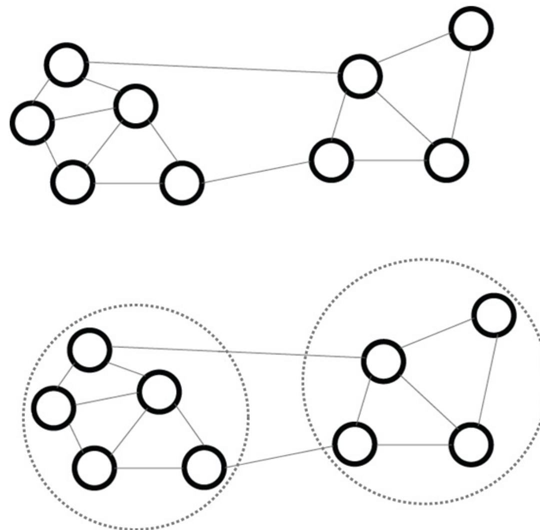


Figure 3.1:

Alexander's representation of the design problem as a network.

The process of creating (synthesizing) or understanding (analyzing) the architecture of a system often follows a process of decomposition. The basic expectations and assumptions underlying decomposition are that (1) each part by itself is easier to grasp and understand, and (2) understanding the behavior and interaction of individual parts can lead to a better understanding of the system behavior as a whole. Whether we are synthesizing or analyzing a system, decomposition can provide several useful design views and perspectives.

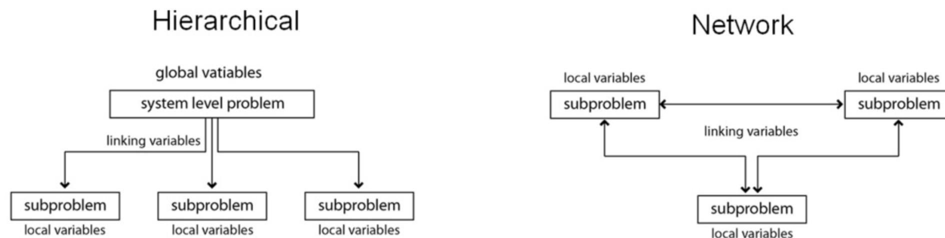
Considerable literature exists on the subject of decomposition. The principal research in this literature was initiated with the work of Alexander (1964) on using networks for representing and decomposing design problems according to customer needs. Figure 3.1 shows a network representation of a design problem.

In this representation, designs are decomposed (or partitioned) into minimally coupled groups. Vertices denote functional requirements while edges denote interactions between them. The interaction strength between functional requirements is inversely proportional to edge lengths. Groups of connected functional requirements represent sub-problems that are relatively independent of other functional requirements and consequently other sub-problems. Clustering in this manner allows for a representation of higher interaction within groups and lower interaction between groups. The individual clusters containing smaller and relatively independent sub-problems can then be solved with minimal effect on the rest of the design.

The two main system structures discussed in chapter two, hierarchical and network structures, are used in decomposition. In a hierarchical decomposition the structure normally transitions from general at the top to specific at the bottom. It continues into finer and finer levels of detail until the lower levels become clearly defined. A network decomposition on the other hand, comprises sub-problems of analogous hierarchies that are directly linked to each other. Figure 3.2 shows an example of hierarchical and network decompositions.

Figure 3.2:

Hierarchical and Network decompositions.



There are several themes along which a design can be decomposed. In the previous chapter I discussed how design can be considered as an object as well as a process. My focus in this chapter will be to represent themes of decomposition that address these two visions. As an Object, design can be decomposed into form, function and discipline aspects. As a process, design can be decomposed into development stages, as well as design activities. In the rest of this

chapter, and building on research in design science and system engineering, among other fields, I will show models of decomposition for each of these themes in some detail, focusing on the inherent processes that occur within them. The discussed models will provide a foundation for the MDDS framework that will be discussed later on.

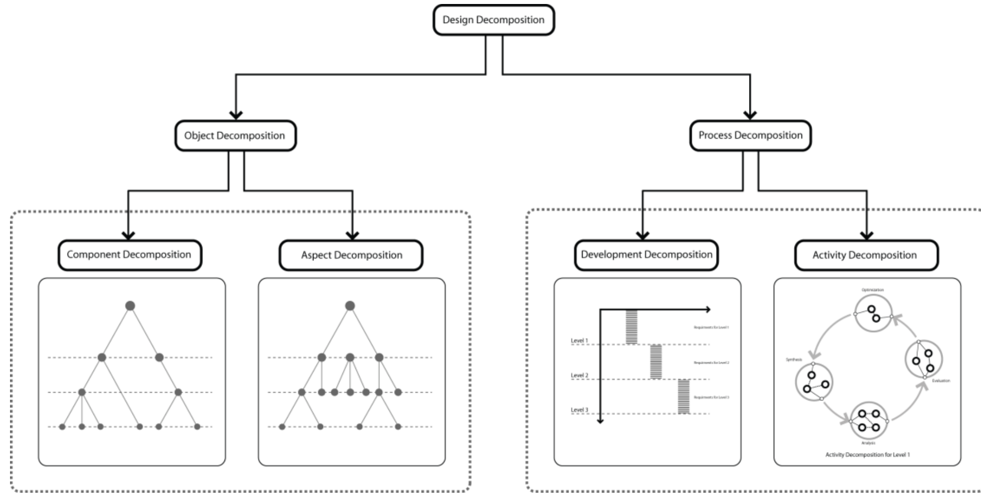


Figure 3.3:

Design decomposition can consist of object and process decompositions. Object decomposition includes Component and Aspect decompositions, while Process decomposition includes Development and Activity decompositions

3.2. Design Object Decomposition

Object decomposition refers to hierarchically related modules, where each module represents a subsystem, presented schematically as a pyramid whose top is the higher-level system and base is the lower level subsystem. Subsystems may correspond to physical components, and in this case the decomposition is called *component decomposition*. Subsystems can also correspond to functions and the engineering disciplines which contribute to the system design. In this case the decomposition is referred to as *aspect decomposition*.

The Axiomatic approach discussed in the second chapter, which attempts to map functional requirements to physical components, is a good design model that describes object decomposition. Object decomposition typically occurs in a top-level fashion, where the system's required functions are broken down into subfunctions. In parallel the system in its physical form is broken down into subsystems that can perform the subfunctions. Decomposition continues similarly until it reaches single parts. Throughout the process, design and testing of physical components are assigned to

different disciplinary parties. This facilitates the synchronized development of different parts of the product by these parties.

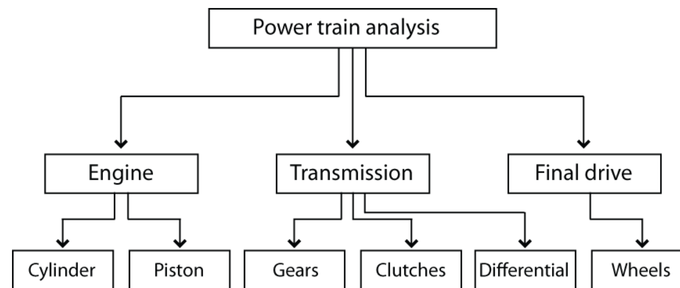
It should be noted however that in system synthesis and analysis, it is not typical that designers persistently follow a top-down decomposition process to the level of single parts. They can also iterate between upper and lower system decomposition levels according to what they can potentially learn within the process about the implications of some of their architectural decisions.

3.2.1. Component/Physical Decomposition

Component decomposition breaks down the problem in relation to the known physical parts (or components). The hierarchy of this breakdown is such that a product's physical elements are organized usually into physical building blocks called chunks. Chunks consist of a group of components that execute the functions required for the product, as illustrated in figure 3.4.

Figure 3.4:

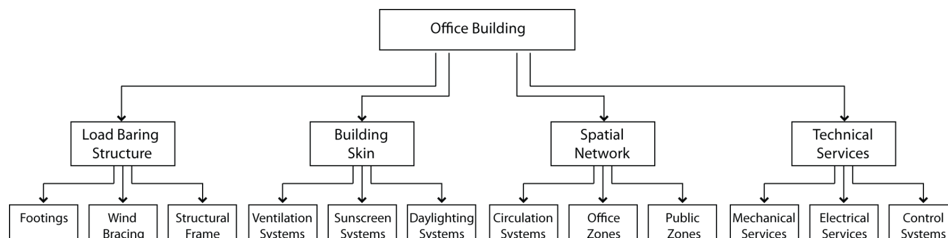
Power train component decomposition.



Some physical elements become more defined, usually with design progress, while others are dictated by the product concept. The outcome of this kind of decomposition is affected by the themes selected for component decomposition, which in turn are influenced by the desired functions that should be performed. Figure 3.5 shows an example of component decomposition for an office building.

Figure 3.5:

Office building component decomposition.

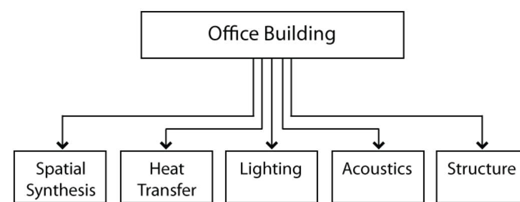


3.2.2. Design Aspects Decomposition

Aspect decomposition (also known as disciplinary decomposition) involves breaking down a problem according to its functions, which in turn can be assigned to a specific discipline that can handle the different physics of the system.
The act of decomposing in aspect decomposition is oriented towards the different domains of knowledge involved in the design problem formulation rather than the physical components. Synthesis and analysis of systems are implemented according to specialties by decomposing the root node which is populated by a high-level function that in turn fulfills lower level technical functions and requirements.

Figure 3.6:

Office building aspect decomposition.



Aspect decomposition usually divides the design system into well-defined categories. For example, automobile design is often divided into a power train team, an interior team, or a ride quality team, etc.

Figure 3.6 shows an example of aspect decomposition in an office building. Several functional-aspects affect the system and are the subject of evaluation (e.g. lighting, air distribution, heat transfer, structural analysis, etc).

Aspect decomposition, however, may fail to account for disciplinary coupling, despite the fact that data exchange may be involved.

In practice, both component and aspect decomposition (or partitioning) are typically used interchangeably. This is often done in an ad hoc manner and is not systematically generated in order to reduce coupling or partition elimination.

3.3. Design Process Decomposition

3.3.1. Design Development Decomposition

By design development, I refer to the complex process which involves the evolution of new designs of systems, artifacts, projects and products over time. This process starts from the moment a need for new designs is recognized and a feasible technical approach is

identified, and continues through developing and introducing the product as a well-formed design.

The need to break down this process originates from many factors debated among a variety of sources. Kossiakoff and Sweet (2002) refer to the large commitments of resources required increasingly throughout design progress. They also refer to inevitable risks which must be identified and resolved as early as possible. Therefore, by decomposing the design development process, the design evolution follows a step-by-step approach. In this approach, the success of each step is demonstrated, and the basis for the next one validated, before decisions are made to proceed to the next phase.

3.3.1.1. Design Development Models

Design researchers have proposed several models to represent design development decomposition. According to Eggert (2004), a design evolves through phases from the identification of customer needs to the realization of a detailed design.

Kossiakoff and Sweet (2002) define the development stages as the concept development stage, the preliminary development stage, and the detailed development stage. In this definition, a design concept that is perceived to best satisfy a valid need is initially formulated and defined. Through a process of continuous development, the concept is finally translated into a validated physical system design meeting the operational, cost, and schedule requirements.

Pahl and Beitz (Pahl and Beitz, 1996) proposed four phases that differ slightly from those proposed by Kossiakoff and Sweet (2002). For the first phase, they introduce task clarification, which deals with design constraints and gathering information about the requirements that need to be embodied in the design solution. They define the concept design phase as that where function structures are established and solution principles are developed to identify concept variants. The embodiment design phase then involves form determination and developing a product in accordance with technical and economic considerations. Finally, the detailed design phase is concerned with laying out materials, surface properties, dimensions, form and arrangement. This phase also involves re-checking economic and technical feasibility, and generating all drawings and other production and specifications documents.

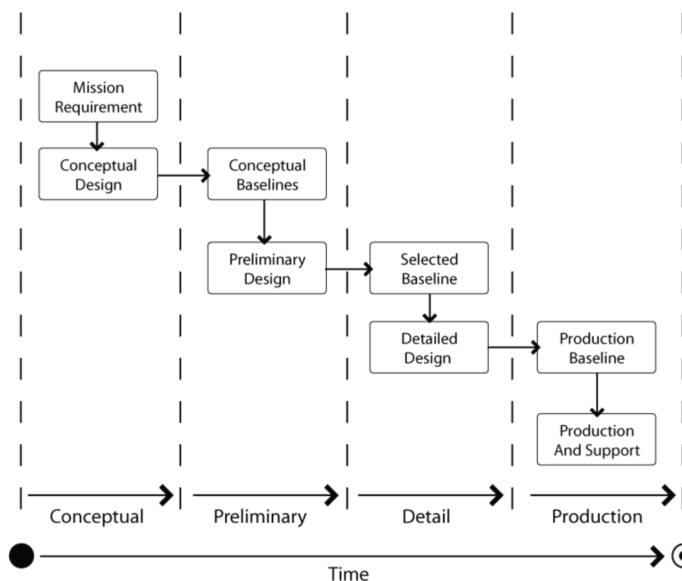
Some systematic approaches to the development process were distilled into specific guidelines. VDI 2221 (VDI, 1985) is a guideline which has attempted to encapsulate the available methodologies into a working framework. Similar to Pahl and Beitz, VDI 2221

suggests four main phases for design development: task clarification, conceptual design, embodiment design and detailed design. Other engineering design researchers extended the embodiment design phase. Dixon and Poli (1995) split it into two phases: configuration and parametric design. Dieter (2000) included product architecture as an additional phase before configuration design.

By looking at the RIBA handbook (1965), a large “plan of work” shows 12 strategies in the design development process. These strategies, described as logical courses of action, include inception, feasibility, outline proposals, scheme design, detail design, production information, bills of quantities, tender action, project planning, operations on site, completion, and feedback. Another simplified version which uses “usual terminology” reduces these into briefing, sketch plans, working drawings, and site operations.

Figure 3.7:

Design development can be decomposed into several stages.



Although all development phases define a sequence of self-contained processes, there should be a reasonable “overlap” and transition at the boundaries of each phase when it comes to real practice. At every step, a decision has to be made as to whether “iterative cycles” of these steps should be carried out in order to proceed to the next activity within the process. (Black, 1990).

Based on the development models discussed above, I will assume that design development is decomposed into conceptual, preliminary, and detailed design, followed by a phase of manufacturing and production (figure 3.7). In the following sections I will discuss these main development stages in more detail.

Concept Development

The concept development stage involves the necessary analysis and planning to understand the needs or requirements for a certain product and the ultimate system foreseen to fulfill those requirements. Several approaches have been discussed to identify the detailed nature of this stage (Sydenham, 2003; Kossiakoff and Sweet, 2002; Eggert, 2004).

The general approach in this stage involves customer or client requirements. These requirements are translated into functions that should be satisfied by the system or artifact. These functions are then synthesized by a team of design specialists into one or more design concepts. Several design alternatives or scenarios are generated in order to select the best alternative for further development. Lower level functional requirements are then assigned to specific system components (Ref07).

Kossiakoff and Sweet (2002) expand this approach to embrace three main subdivisions: the analysis phase, the concept exploration phase, and the concept definition phase. In the analysis phase, the basic needs and requirements for a new product are defined in a continuous search for a "practical approach" that can possibly satisfy those requirements. The concept exploration phase tends to formulate and validate specific performance requirements for a set of potential proposed concepts. This phase focuses on how these performance measures address the original requirements and sets a valid goal for a new product. This is done for all potential concepts before exerting major effort on individual development. The concept definition phase looks at key characteristics of the alternative concepts and selects the most beneficial in terms of performance, estimated cost, development and operational life. After defining the functional characteristics of the preferred concept, major resources are committed to carry this concept forward to subsequent phases of preliminary and detailed development.

Eggert (2004) focuses on problem formulation as the key activity in the concept development phase. In this approach, greater attention is dedicated to understanding the problem, exploring the stated needs of the customer, clarifying the expected system performance and determining required disciplines. During concept design, the performance of alternative concepts or working principles is evaluated using simple calculations or physics relations in order to choose the best candidate by means of a set of evaluation criteria.

This process of design concept synthesis is considered the most creative part in the evolution of a design. This is where the designer

creates a new concept by use of an impulsive synthesis of intuition and previous knowledge, depending on special skills and experience.

Designers usually view the initial concept design activity as intuitive rather than scientific. There is thus very little communication between designers at this point, which makes it difficult to extract from them any organized information.

In order to arrive at a satisfactory concept, an iterative process is sometimes required. This includes identifying the overall design objectives, defining a concept, gathering data for assigning model parameters and design vectors if possible, and gathering information on the system structure and operating procedures (Lawson, 2005). It is never enough to have one single person or document to perform these operations.

Sydenham (2003) describes some of the methods that are used to set a basis of comparison for alternative concepts so that their distinguished features may be scrutinized. Some of these include mind-maps, rough CAD models, systems dynamics, scenario building, and other motivational modeling methods. Sydenham (2003) states that development along the concept design phase is best performed with top-down thinking initiated by the customer requirements, but also informed by some bottom-up knowledge to maintain practical possibility.

Preliminary/Embodiment Development

Although it appears to be easy for design practitioners to define task clarification, concept design or detailed design, the definition and location of the preliminary design, also known as embodiment design, in the overall process structure is not clearly defined (Pugh and Morley, 1989).

The significance of the embodiment phase arises from its being able to bridge between conceptual and detailed activities. The crude nature of concept design models does not allow for a comprehensive evaluation in terms of cost, time and performance practicality. The main contribution of the embodiment phase is the ability to assess the feasibility of a candidate design from both an integration and implementation point of view (Sydenham, 2003). In this context, a better and more realistic understanding about the realization of candidate solutions and their practical limitations should be achieved. The end result of this understanding should be a definitive output, providing a selected candidate that seems outstandingly promising.

According to Black (1990), solution concepts are translated within the embodiment stage into geometrically precise “layouts” that are representational models of the product configuration. This configuration contains all the necessary information in terms of geometry, material, position and topology for its subsystems and components. This allows for the development of a technical system that satisfies the requirements of functionality, constructability, cost and other factors (Hubka et al., 1988).

Eggert (2004) offers a broader definition of embodiment design, which refers to configuration design as well as parametric design. The configuration design phase is where the alternative layouts and configurations are generated, analyzed and evaluated against technical and economic criteria in order to select a best candidate. Parametric design, however, involves defining values for controllable parameters and design variables for the configuration, shape, size and material of the design. Using formulas, experiments and computer programs, the performance of these designs is analyzed, and the analysis results are checked to ensure acceptable performance and constraint satisfaction. Otherwise, new alternatives are generated with new design variables for another reiteration of analysis and evaluation, and so on.

Detailed Development

The main bulk of the process of engineering the system to satisfy the functions and requirements specified earlier in the concept and embodiment phases lies in the detailed design development stage. In this phase, specialist area engineering designers acquire maximum knowledge about the object of design while developing the actual nuts and bolts decisions that allow its physical formation (Sydenham, 2003).

This increased knowledge is expressed in the form of highly elaborated packages of manufacturing specifications and assembly procedures. These include detail and assembly drawings, bills of materials, manufacturing process recommendations, and prototype performance test results (Eggert, 2004). In addition, product specifications are also key constituents of this phase, such as height, width, depth, weight and expected performance. All these forms of design output prescribe the physical features of the assembled parts that generate the required system when fabricated.

Here, design freedom is at its minimum, where it becomes extremely expensive and time-consuming to correct any errors or to modify any features in the design (Sydenham, 2003). The output of this stage is usually irreversible. The project is transformed from the paper or

computer model phase to a "cut metal" commitment. There is no time to lose on modifications if some features are wrong, and the output is otherwise sent to scrap.

The detailed development design phase is where all issues of reliability, constructability and maintainability, hinted to in earlier phases, are of highest priority. This phase manages the engineering change process to maintain configuration and interface control. It also manages the integration and testing of the product components to function within the system. At the individual component level, this phase also guarantees the reliable implementation of all functionality and compatibility requirements.

The detailed development design phase involves the realization of an integrated complex system as a whole consisting of engineered components, as well as the evaluation and testing of the system operation in a real environment.

3.3.1.2. Proposed Design Development Model

The design development model proposed in this thesis is concerned with setting certain requirements, deliverables, targets and milestones for each stage in the design evolution and development.

The design development model can be decomposed into the three design stages mentioned earlier, or it can be divided into many more. For example, the three main development stages proposed can each be decomposed into several other sub-stages.

It should be noted, however, that the design development decomposition is a description not of the process but of the required products of that process. It tells us not how the design team works but, what must be produced in each stage. Further, it also details the services provided by the design team and therefore can be used to determine agreed stages of work which could attract staged payments. So the plan of work may also be seen as part of a business transaction; it tells clients what they will get, and describes what the design team must do. It does not necessarily tell us how it is done (Lawson, 2005). This will be discussed in the following section.

3.3.2. Design Activity Decomposition

In this section, I propose a model of design activity decomposition. In order to do so, I will first discuss the definition of design activities and their evolution through tracing and reviewing previous models of design activities.

3.3.2.1. Design Activity Models

Several models and attempts were made during the late 1950s and the 1960s to describe the creative problem-solving process in design by means of a structured series of phases that define dominant activities, such as synthesis, analysis, evaluation, and so on (Rowe, 1987). Some of these models *prescribe* what they perceive as a better or more suitable pattern of activities and are called prescriptive models of design. Others tend to simply *describe* the sequences of activities that typically occur in the design process and are thus called descriptive models. (Cross, 1989).

The primary concern of prescriptive models of design is encouraging designers to adopt a specific design methodology while working. This is usually in the form of algorithmic or systematic procedures to follow.

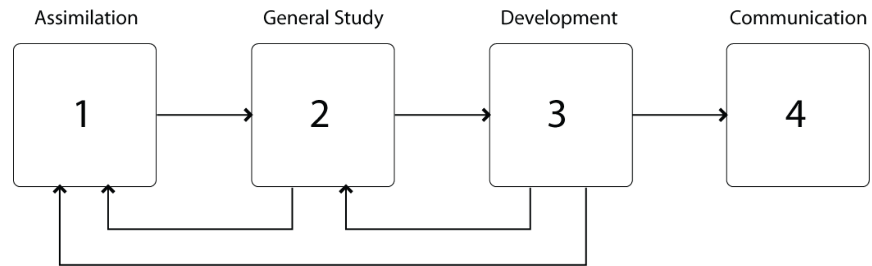
On the other hand, the descriptive models of the design process simply describe the conventional "heuristic" process of design. In this approach, the primary focus is on the *synthesis* process in order to come up with a solution early on, thus reflecting the solution-focused nature of design thinking. This preliminary solution is then followed by *analysis* against design goals and constraints, and then consequently *evaluation,* refinement and *optimization*. The analysis processes usually point out basic problems in the preliminary solution, and so it has to be replaced with another solution, which is synthesized and the loop goes on. The endpoint of this process is the communication of a new design.

RIBA's model

The RIBA *Architectural Practice and Management Handbook* (1965) suggests a prescriptive model in which the design process is divided into four phases: assimilation, general study, development, and communication, (shown in figure 3.8). The assimilation phase deals with the accumulation and ordering of information related to the design problem. The general study phase investigates the nature of the problem, while exploring means of possible solutions. The development phase involves developing and refining one or more of the candidate solutions outlined during the previous phase. The final phase communicates one or more candidate solutions to people inside or outside the design team.

Figure 3.8:

RIBA's four phase model which includes: assimilation, general study, development, and communication.



Although it may seem from the logically sequential nature of the diagram that these phases progress smoothly in the same manner, a closer reading reveals quite a different scenario. In reality, there is a continuous interaction back and forth between most of these phases. A designer can rarely know what information to collect in the assimilation phase unless some investigation of the nature of the problem is done in the general study phase (Lawson, 2005).

Moreover, development and refinement does not ideally progress into one solution. Sometimes more detailed refinement requires that the designer go back to better understand the problem and gather other relevant or unconsidered information in the first place. Another common scenario could even take the designer from the final phase back to square one, where presenting a fully implemented design solution to a client forces the client to go back and describe the problem again more clearly.

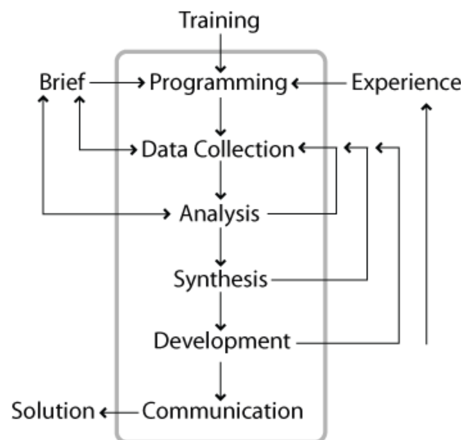
Analyzing this type of design map could reveal innumerable similar scenarios. The bottom line, however, is that the designer has to collect information, investigate the problem, develop a solution and implement it for communication purposes. These activities need not be done in that specific order. There can be unpredictable loops among these activities, but this model does not specify the nature or frequency of these loops.

Archer's model

Archer (Archer, 1984) developed a prescriptive model which focuses on interactions with the world outside the design process, such as client requirements and inputs, the designer experience and training, and other sources of information. The output in this model is the communication of a particular solution. These inputs and outputs to and from the design process are shown as external to the design process in the flow diagram in figure 3.9, which also exhibits many feedback loops.

Figure 3.9:

Archer's model includes six types of design activity: programming, data collection, analysis, synthesis, development and communication.



Archer identified six types of design activity within this model: programming, data collection, analysis, synthesis, development and communication. *Programming* involves the establishment of fundamental issues and proposes a main course of action. Through *data collection*, classification and storage is achieved. *Analysis* is then performed to identify sub-problems, prepare design performance specifications and reappraise proposed programs and estimates. The *synthesis* process proceeds to prepare outline design proposals, which are *developed* into prototype designs and prepared for validation studies. Finally, the design communication process follows through preparing manufacturing documentation.

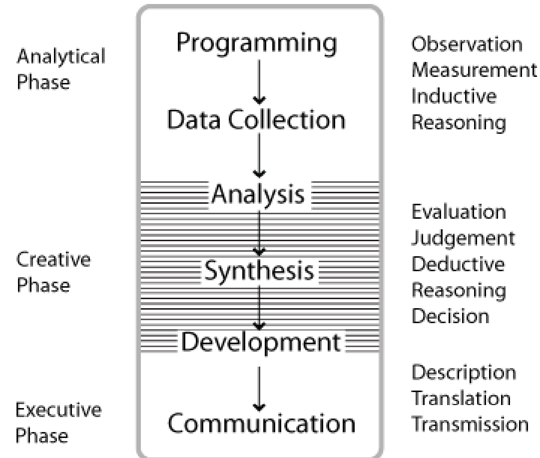
For Archer design was a sequence of identifiable activities that occur in a logical and predictable order and are defined by the type of task involved. Although the activities are identifiable, the phasing is less discretely defined than in the RIBA Model owing to the strong feedback loops and relationships between activities. Archer suggested three interconnected domains within this process: external representation, process of activities, and the problem solver. He therefore demonstrated a distinction between explicit behavior and the cognitive realm, where the emphasis always remains on the explicit behavior apparent in the sequence of activities.

Archer reduced these activities by dividing the design process into three broad phases: analytical, creative and executive phases, (as shown in figure 3.10). The analytical phase requires objective observation and inductive reasoning, while the creative phase, the heart of the process, requires involvement, subjective judgment, and deductive reasoning. After making most of the important decisions, the design process evolves into an execution phase, which involves

the objective and descriptive production of working drawings, schedules, etc. The design process is described in this context to be a "creative sandwich", where the creative act lies always in the middle between layers, thick or thin, of objective and systematic analysis. This model therefore suggests a basic structure of synthesis-analysis-evaluation-refinement.

Figure 3.10:

Archer's reduced model with three broad phases: analytical, creative and executive phases.



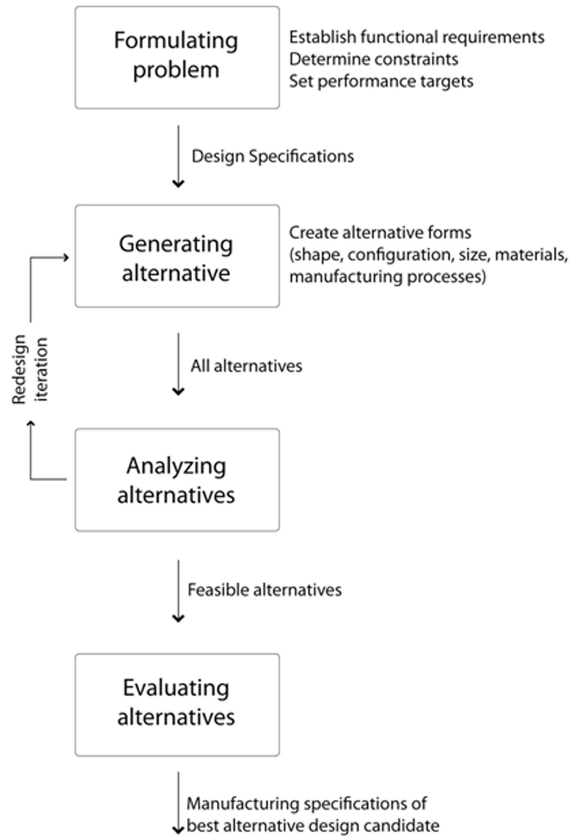
Eggert Design Model

Eggert's (2004) model, as shown in figure 3.11, describes four basic phases: formulating, generating, analyzing and evaluating. The formulation phase includes all activities and decision-making processes implemented in order to understand design problems and plan their solutions. In this phase, information is gathered regarding customer needs and required performance. Constraints are also set to determine economic, technical, legal and safety considerations. In this phase, detailed engineering design specifications are developed to guide decisions downstream.

The generating phase describes the activities and decision-making processes used to create candidate designs to be scrutinized later in the analysis and evaluation phases. The methods used to generate such candidates at the concept design level can include creative methods, such as brainstorming and Synectics. The generation process then progresses into more developed stages, with more defined layouts, configurations, shapes, sizes, materials, or manufacturing processes.

Figure 3.11:

Eggert's design model includes four basic phases: formulating, generating, analyzing, and evaluating.



The analyzing phase is concerned with predicting the behavior of a design candidate. This is accomplished by preparing engineering models using knowledge from the basic sciences and computational skills from mathematics to predict the performance of each candidate design. This phase determines if the design should continue into developing phases or if a reiteration is required. This reiteration implies that new values for the design form are selected, and then the redesigned candidate is consequently reanalyzed. This is illustrated by the solid reiteration loop in figure 3.11. The design problem may require a complete redefinition of specification or constraints if no appropriate candidates are satisfactory.

The evaluating phase is concerned with comparing the predicted performance of each "working" design candidate to determine the "best" or optimum design alternative. The evaluation criteria, which exist in the engineering design specifications, include performance measures such as speed, size, reliability, maintenance intervals, power, weight, and cost. New candidate designs can be automatically regenerated using some computerized numerical techniques in order to enhance overall quality and performance.

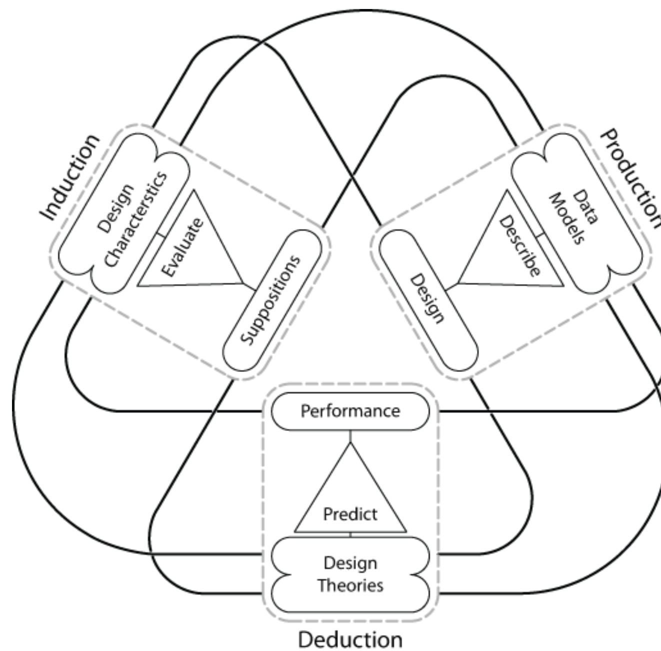
In engineering design problems, the value or weight of some evaluation criteria may be more significant than others (e.g. strength vs. cost). This requires embedding decision-making methods that exhibit compromises of candidate solutions, where some performance measures can be promoted while the others are degraded. In this case, these *logical* methods, as in the analysis process, will replace numerical equation solving.

March's PDI model

March (1984) suggested a design process that deals with the solution-focused nature of design thinking. As shown in figure 3.12, March argued that the two conventionally understood forms of reasoning - inductive and deductive - only apply logically to the evaluative and analytical types of activity in design. He considered synthesis as the type of activity that is most specifically associated with design, as it does not entail any commonly acknowledged form of reasoning.

Figure 3.12:

March's PDI production, deduction, induction model.



This model drew on the work of the philosopher Peirce to identify this missing concept of "abductive reasoning". Deduction involves proving that something "*must be*"; induction implies that something is "*operative*"; while abduction suggests the fact that something "*may be*". In this hypothesis, it is clear that synthesis, referred to as what "may be", is the act significant for the process of designing, since it is concerned with generation or production. March thus

coined such a reasoning process as "productive reasoning". His model for a rational design process was therefore called the PDI (production – deduction – induction) model.

In this model, the first phase, productive reasoning, draws on a preliminary statement of requirements and some presuppositions about solution types in order to produce or describe a design proposal. From this proposal and established theory (e.g. engineering science) it is possible to deduce or predict the performance of the design. From these predicted performance characteristics, it is possible to evaluate further suppositions or possibilities through induction, leading to changes or refinements in the design proposal.

The Function Behavior Structure Framework

The function–behavior–structure (FBS) framework by Gero (1990) introduces an important formal representation of the design process. Three main classes of variables describe aspects of a design object within this framework: function (F), behavior (B) and structure (S). (F) variables describe what the object is for, or the teleology of the object. (B) variables describe what the object does, in the form of the attributes that are derived or expected to be derived from the object's (S) variables. (S) variables describe what the object is, in terms of its components and their relationships.

The FBS framework thus represents the act of designing by a group of processes which link function, behavior and structure together, as illustrated in figure 3.13. These three aspects are viewed in this framework as different states of the developing design.

In the general context of designing, a function F (where F is a set) is transformed into a design description (D) of a particular artifact that can produce this function. The design description takes the form of drawings in this case. A preliminary model of this design is denoted by: F → D, where → is a transformation. However, no direct transformation is capable of attaining this result. Structure (S) represents the elements of the artifact and their relationships.

Another activity of design is F → S. Similarly, no direct transformation between function and structure exists. This therefore requires an indirect transformation between function and structure. Bobrow (Bobrow, 1984) has defined function as the relation between goals of a human user and system behavior. In the design process, behavior is regarded in two ways. The behavior of the structure (Bs), which is a process of analysis that marks out which behaviors to determine, is directly derivable from structure

according to the relation: S → Bs.

The other view of behavior in the design process is concerned with transforming function to expected behaviors (Be). The expected behavior provides the syntax by which the semantics represented by function can be achieved: F → Be.

The predicted behavior of the structure (Bs) can be compared with the expected behavior (Be) which is required to determine if the synthesized structure can produce the functions, according to the relation: Be ←→ Bs, where ←→ is a comparison.

Another model of design is F → B, Be → S(Bs). Here, the function is transformed to expected behavior. The expected behavior is then used to select the design artifact structure based on knowledge of the behaviors produced by this structure. Finally, through some drafting tools, structure can be transformed into a design description, represented by the relation: S → D.

Therefore, within this framework, the designer creates associations and relationships between these three states through experience. Function is assigned to behavior, while behavior is derived from the object structure. As mentioned earlier, there is no direct connection between function and structure.

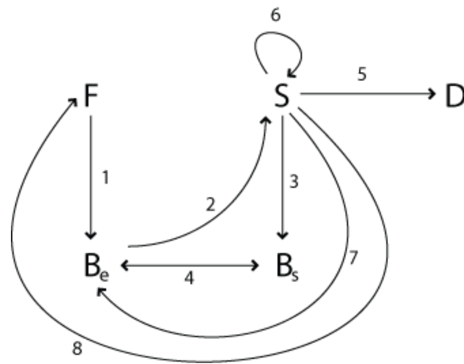
Gero highlights eight processes in the FBS framework as shown in the diagram, arguing that they are fundamental for all designing. Process 1 (formulation) transforms the design requirements, expressed in function (F), into the expected behavior to enable that function (Be). Process 2 (synthesis) transforms the expected behavior (Be) into a solution structure (S) to exhibit that required behavior. Process 3 (analysis) extracts the actual behavior (Bs) from the previously synthesized structure (S). Process 4 (evaluation) compares (Bs) with (Be) to generate decisions regarding accepting or rejecting the proposed design solution. Process 5 (documentation) generates the design description (D) for product manufacturing.

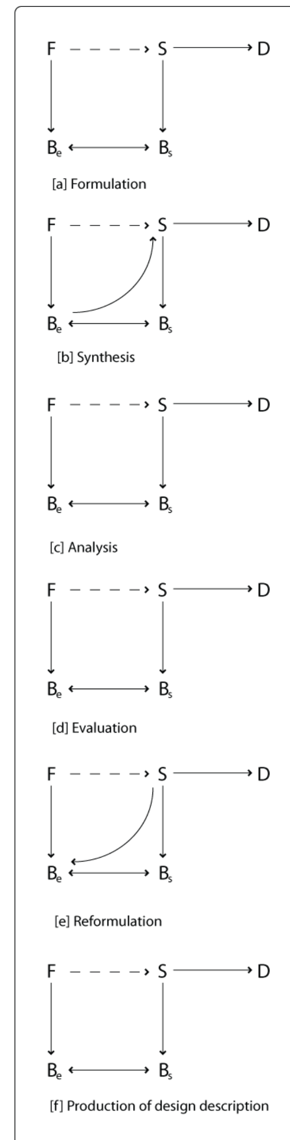
Processes 6, 7 and 8 (reformulation types 1, 2 and 3) are the most significant types of processes that do not appear in most conventional design models. They reflect a different and non-static view of the world of designing in the FBS framework. Reformulation occurs usually when the behaviors produced by specific structures alter the range of expected behaviors and consequently the initial functions. It can also happen due to an evaluation which yields an unsatisfactory relation between (Bs) and (Be), which at the same time cannot be made satisfactory by changing the structure. A

change in expected behavior thus occurs in this case. Reformulation type 1 is the most explored process, however, as it is evident in examples like case-based reasoning (7) and structure analogy (8). Some empirical design studies (9) confirm that the reformulation type 1 is the prevalent type, while the activity of reformulation in types 2 and 3 diminishes but does not disappear during the design process.

Figure 3.13:

The Function–Behavior–structure (FBS) Framework (Gero , 1990)





3.3.2.2. Proposed Activity Decomposition Model

Based on the several activity models presented, I will propose a simple design activity model that will be the basis for the design activity decomposition within this thesis. This model will be composed of synthesis, analysis, evaluation and optimization design activities and will be organized in a cycle.

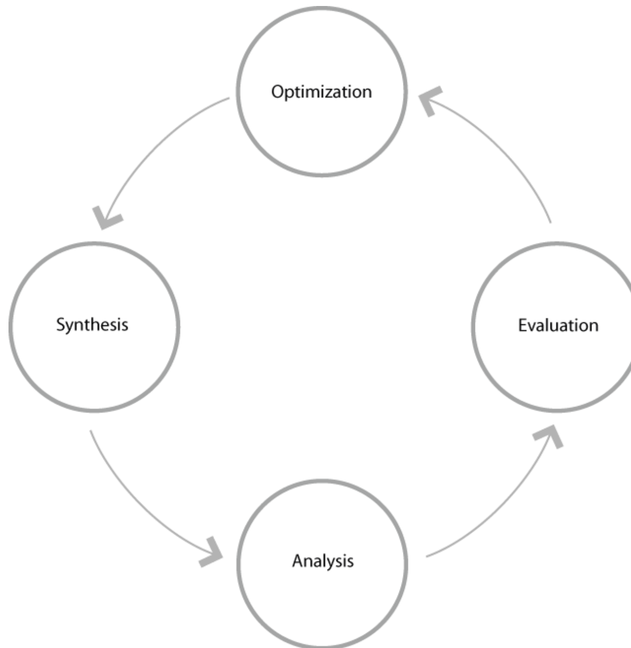


Figure 3.14:

Design activity model. Four phases are included: Synthesis, Analysis, Evaluation, and Optimization.

The synthesis and analysis design activities are similar to the activities discussed in this chapter and in the previous chapter. The optimization activity is based on Minsky's Progress Principle mentioned in chapter two. This activity helps guide the design generation.

This optimization activity is easy to understand if the design has only one objective since progress then simply implies making that objective better. But when there are many different or even conflicting objectives, progress becomes harder to define. An evaluation activity is needed to handle the decision process in such design problems and to manage the tradeoffs between the different objectives. Based on this the design activity cycle will include: synthesis, analysis, evaluation, and optimization (figure 3.14).

3.3.3. Hybrid Design Process Models

For reasons of clarity within this thesis, both development and activity process decompositions are treated as two distinct views of

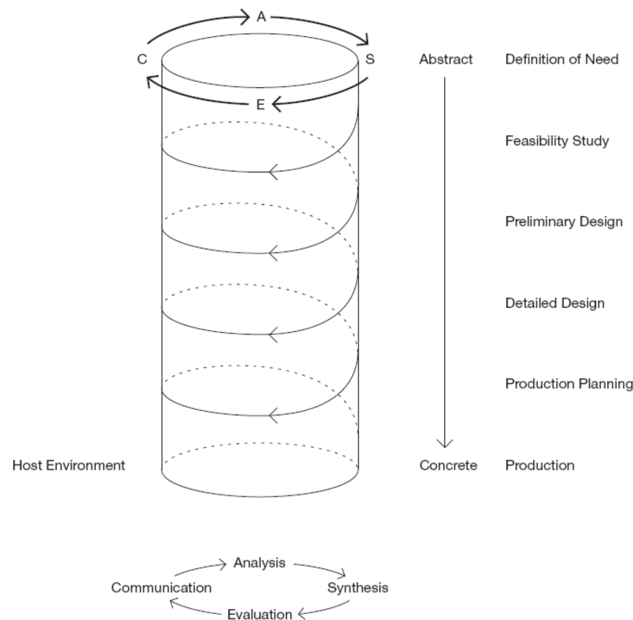
the design process. However, there is a strong relationship between both design processes. Several design models were developed that attempt to capture that relationship.

Asimow's Model

The work of Asimow, an industrial engineer well-known in the 1950s and 1960s, illustrates further contribution to the logical structure of phases of activities within the design process. Asimow (Asimow, 1962) described two structures in the design process: a vertical and a horizontal structure, (as shown in figure 3.15).

Figure 3.15:

Asimow's design model includes a vertical and a horizontal structure. (Mesarovic, 1964)



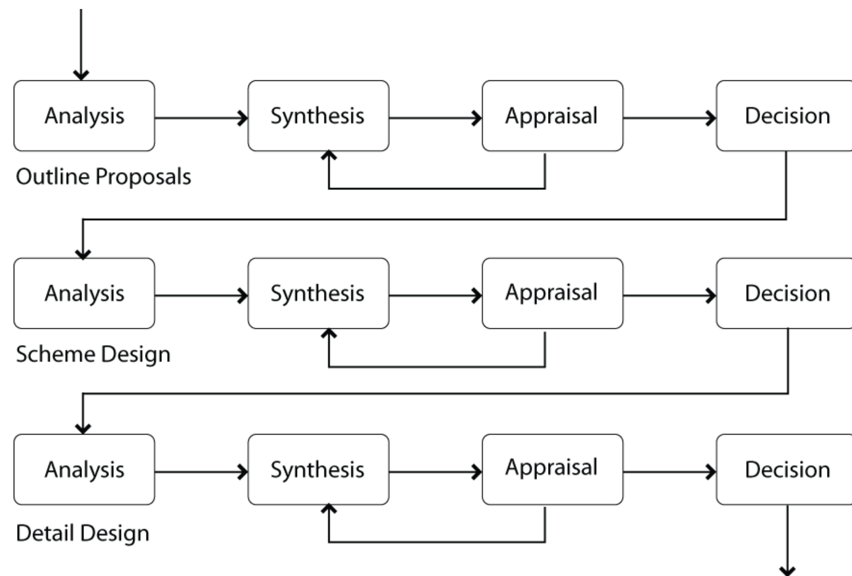
The vertical structure involves a chronological phasing of activities, starting from the definition of needs, moving through feasibility study, preliminary design, detailed design, production planning, and finally production. The overall sequence of activities was viewed by Asimow to move from abstract considerations at the beginning to more concrete and solid ones further on. Feedback loops were integrated between phases for the purpose of tracking information or any problems through the whole process and responding accordingly. Asimow represented the horizontal structure as an iterative decision-making cycle that lies both within and between the various phases of activity. This cycle begins with analysis, and then proceeds through synthesis and evaluation to communication (Rowe, 1987).

Markus and Maver Model

Tom Markus (1969) and Tom Maver (1970) produced maps for an architectural design process, where they argued that a fully integrated map requires both a “decision sequence” and a “design process” or “morphology”. They suggest that the decision sequence consists of the phases of analysis, synthesis, appraisal and decision, which occur at increasingly detailed levels of the design process.

Figure 3.16:

The Markus/Maver model includes a decision sequence and a design process.



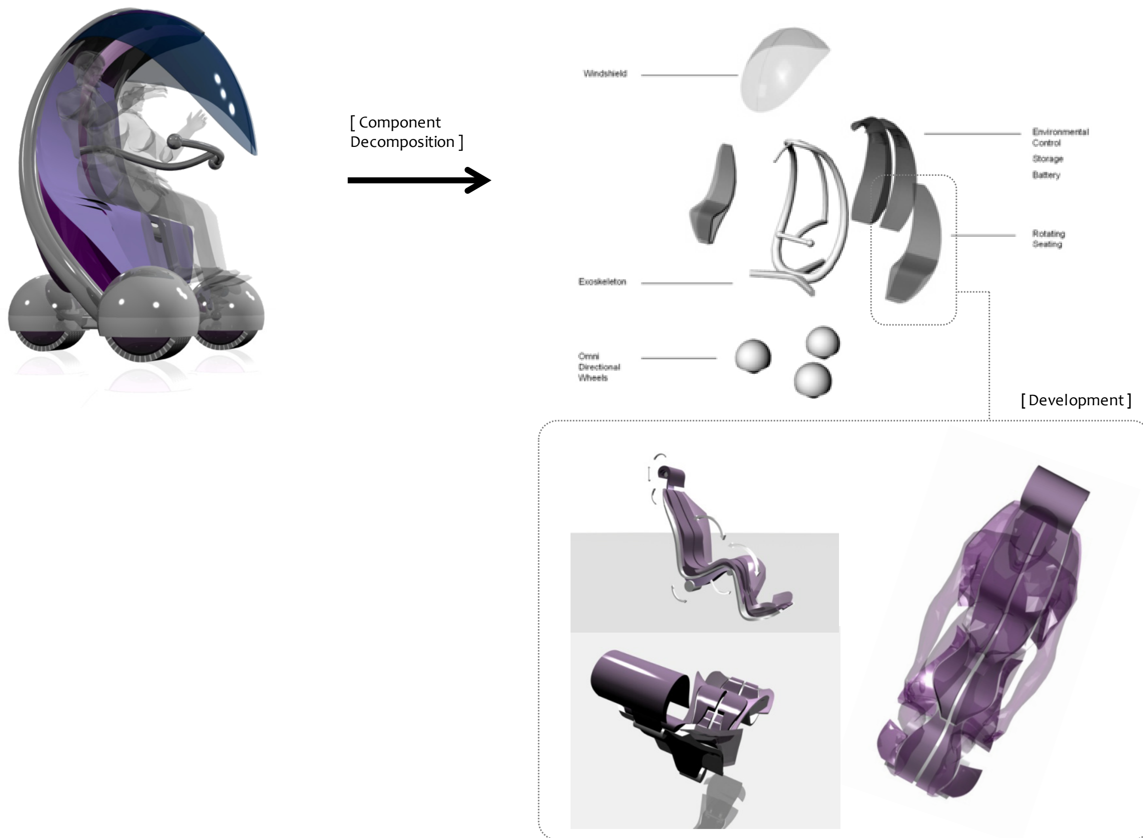
It should be noted that some of the terminology and names used to describe these activities in this model are different in meaning than in other models mentioned earlier, although analogous in their terms. Analysis, for example, implies here the exploration of relationships and patterns of available information, leading to the defining goals and objectives. In this context, it does not map to the same definition of analysis in the scope of my research, as it corresponds to an earlier activity of problem structuring and ordering and information gathering. Synthesis is described in this model as the attempt to progress and generate a response or solution to the problem. Appraisal deals with the critical evaluation of proposed candidate solutions against the goals specified earlier in the so-called analysis phase. The appraisal phase in this sense corresponds to the conventional analysis phase in most of the models mentioned here.

As in most models described earlier, the Markus/Maver model accounts for return loops between activities within the process. For

example, if the designer finds a specific solution he had proposed during the synthesis activity not fulfilling the required goals, he would propose another idea, thus making a return loop in the decision sequence from appraisal to synthesis again (figure 3.16).

Figure 3.17:

The city car project demonstrates that a design can have several decomposed views.

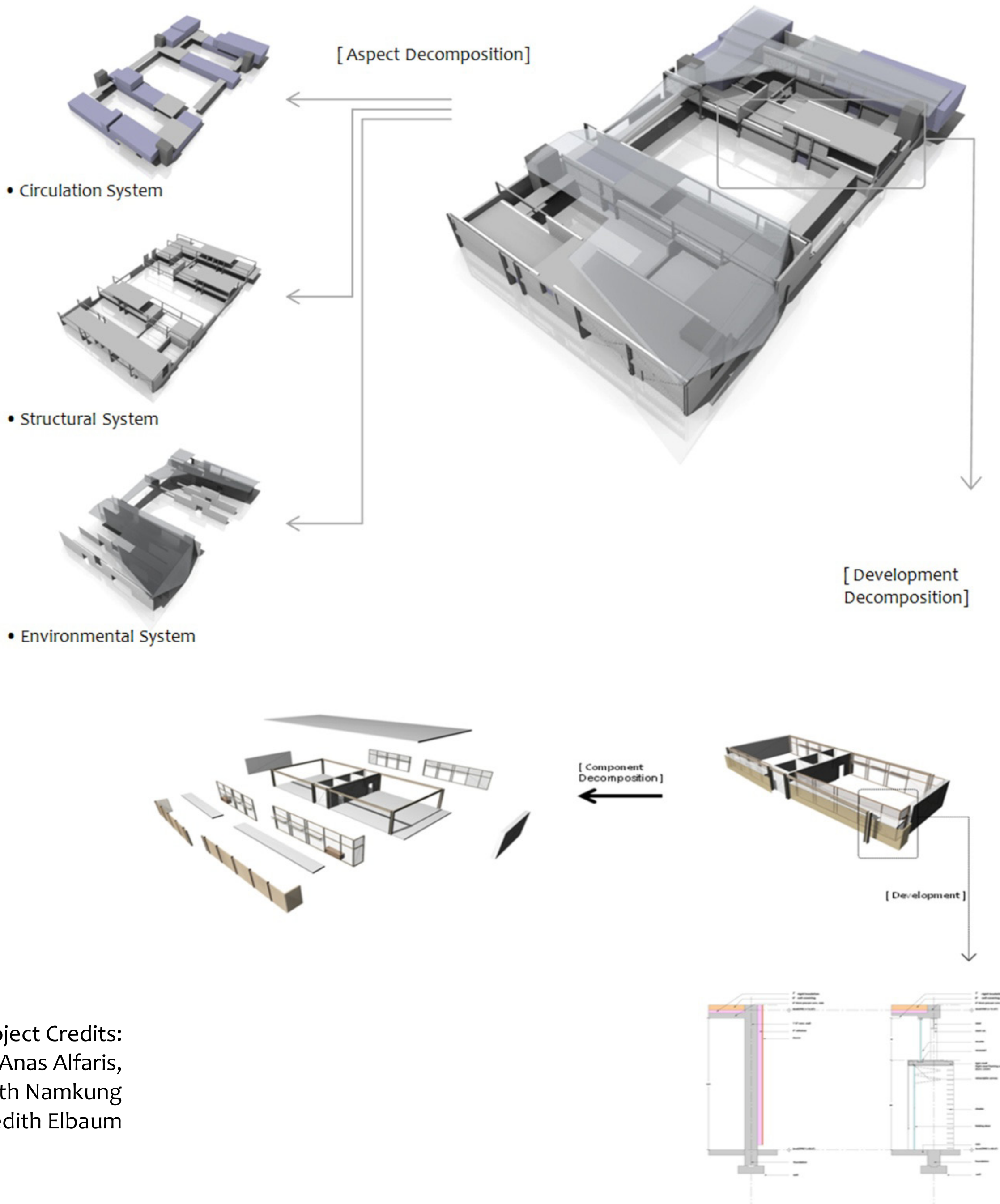


3.3.4. Decomposition and Design Views

It should be noted that there is a relationship between all decomposition views suggested in this chapter. As the design evolves it can be decomposed into anyone of the four different views discussed in this chapter as illustrated in figures 3.17 and 3.18.

Figure 3.18:

In this school project several decomposition views are produced simultaneously.



Project Credits:
Anas Alfaris,
Kenneth Namkung
Meredith Elbaum

4. Formulation

4.1. What is Formulation?

The Webster online dictionary defines formulation as an act of giving form or shape to something or of taking form. It can imply developing something, something that is formed, the way by which a thing is formed, or an arrangement of a group of people or things in a prescribed manner or for a specific purpose.

When synthesizing and analyzing a system, it is hard to identify when decomposition ends and when formulation starts or vice versa. In fact much of the literature on design does not distinguish between both processes. In the context of the MDDS framework, if decomposition is the stage where the designed artifact or system and the processes used to design it are broken down into several components, then the formulation stage is where those components are put together to create the MDDS architecture (figure 4.1).

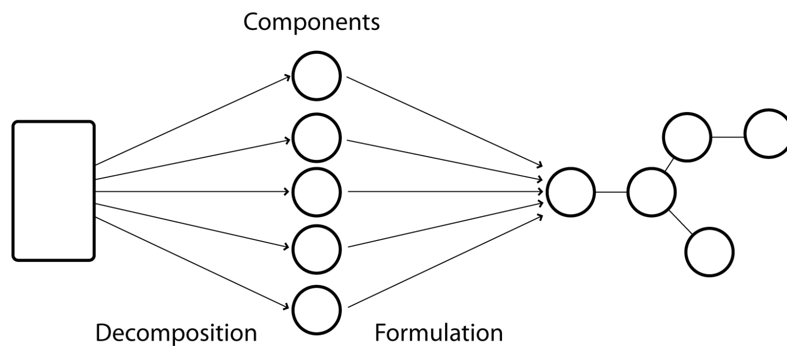


Figure 4.1:

Decomposition breaks a system into components whereas formulation puts them together.

We must distinguish, however, between the physical embodiment, which emphasizes the physical artifact, and the informational and design processes, which are oriented towards the design activities. In this chapter, we will look at both while focusing on the design process in which formulation can be viewed as the process of designing and modeling the design process.

This design process modeling is based on the fact that design processes comprise a number of smaller design activities. The design

process can be modeled by tracing design information exchanged between different design activities.

In addition, at different design phases, there is a need to represent different levels of description of an object or to vary its depth of decomposition. When setting a building within a site for instance, the whole building may be viewed as one single element. Information about details such as the building stories, rooms and spaces becomes irrelevant. A general representation is thus needed to enable shifting between one view and the other in a way that sustains component encapsulation (Rosenman and Simoff, 2001).

Within the MDDS framework, formulation defines the system architecture, describes different degrees of abstraction, and demonstrates how various design activities are going to be connected together through compatible interfaces. An iterative loop should link decomposition and formulation to achieve a reasonable architecture for this process.

Formulation in the context of the MDDS framework would typically take place before mathematical modeling and software programming to avoid major reprogramming later on. It basically promotes the interaction among the system architects, design specialists and other project members, as well as allowing the visualization of control flow and data.

However, with the large number of constituents and increasing complexity of the architecture of individual design activities, the need arises to agree on and adopt a generic formulation modeling language, notation, ontology or meta-model to describe and plan the sequence of applications and interactions, and provide a common basis for all the disciplines and parties involved in the process.

Different tools, notations and methods are needed for the process of creating system structures and architectures. Some notations include software structural analysis and design, while others deal with system engineering build block diagrams or developing data flow diagrams; other kinds of notation involve modeling languages such as UML and SysML. These notations and many others will be discussed briefly in this chapter.

4.2 Process Analysis and Structuring

In the process of designing the MDDS design process, it is necessary to structure the information and different components extracted from the design concept in the decomposition phase.

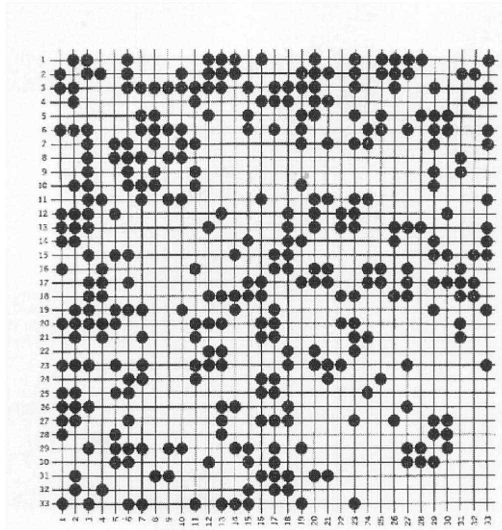
To learn more about how such structuring is accomplished, this section will discuss process analysis and structuring techniques in general, focusing on the design structure matrix (DSM) as a method for complex system structuring.

One of the early attempts to define structural formulation techniques was carried out by Chermayeff and Alexander (Chermayeff and Alexander, 1963) who pointed out that there are structural patterns pertinent to each problem. They suggested that "good design" relies primarily on the ability of the designer to act according to these structural patterns. In order to highlight these patterns, they proposed a method in the early 1960s that implements hierarchical structuring. This method, developed in Alexander's *Notes on the Synthesis of Form* (Alexander, 1964), enumerates and organizes elementary problem statements.

The structure highlighted by Chermayeff and Alexander identifies links between the given problem issues (figure 4.2). The links are defined through designer common sense and experience. The links affect each other through different patterns. The significance in this structure lies in those patterns and not the links as such. The issue of patterns was later developed in Alexander's book *A Pattern Language* (Alexander et al., 1977).

Figure 4.2:

There are structural patterns pertinent to each problem (Chermayeff and Alexander, 1963).

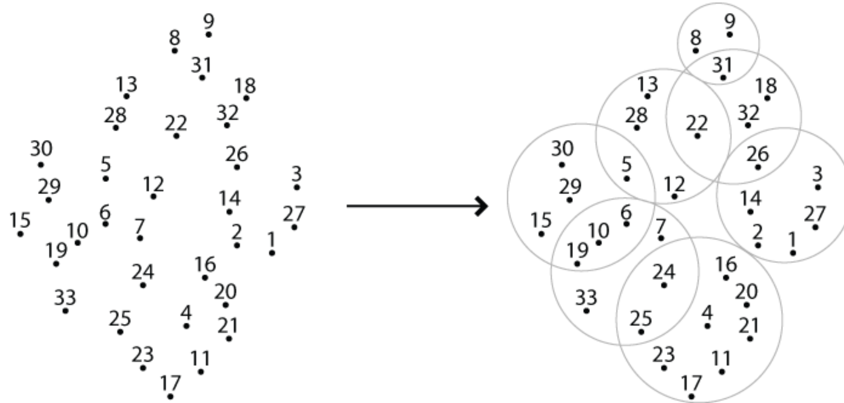


Links between sub-problems are defined in terms of "clusters" or groups of related issues that share many connections, as it is difficult to consider each and every link due to their large number. This grouping becomes significant in the structuring process and is

difficult to achieve. It requires grouping issues that are strongly related together, while considering that elements in different groups need to be significantly independent of one another (figure 4.3).

Figure 4.3:

Issues that share many connections are grouped together (Chermayeff and Alexander, 1963).



Matrix theory has been used in multiple disciplines, such as sciences, mathematics and engineering, to represent systems pertaining to equations, constraints and state variables. It has also been used in design theory within the context of design structure matrices (DSM). DSM, also known as N^2 diagrams, originated from the work of Donald Steward (Steward, 1981), who highlighted the use of matrix-based techniques for analyzing the design structure of systems. DSM is typically used in systems engineering to display component interactions (Grady, 1994).

Figure 4.4:

An activity-based DSM for the development of a soda bottle (McCord 1993).

		A	B	C	D	E	F	G	H	I
Perform Market Research	A	A								
Select Bottle Material	B	■	B	■	■		■		■	■
Design Bottle Shape	C	■	■	C		■	■		■	■
Select Cap Material	D	■	■		D	■		■	■	■
Detail Cap Geometry	E	■		■	■	E		■	■	■
Develop Bottle Mold	F		■	■			F			
Design Cap Mfg. Process	G				■	■		G		
Layout Assembly Process	H		■	■	■	■			H	
Test Cap Sealing	I	■		■		■				I

DEPEND

PROVIDE

Design structure matrices are representations of complex systems that capture system transactions in a simple format and represent the relationships between system components, activities or teams in a highly visual and analytical format. They are thus useful for modeling systems, networks and processes. They are particularly useful for defining activity clusters and tracking interfaces, as they

capture a snapshot of the flow of information. Figure 4.4 illustrates an activity-based DSM for the development of a soda-bottle.

In order to simplify the system interface and enhance the system architecture effectively while preserving the functional conditions, the structure and conventions of the DSM matrix have to be comprehended. A DSM contains identical row and column labels representing the same set of architectural elements. The matrix reveals the interaction of activities. Reading across rows defines the other activities on which a specific activity relies for information; in other words it identifies sources of input. Reading down columns, however, defines which other activities are provided with information from that specific activity; in other words, it identifies output sinks.

For each pair of elements or activities in the matrix, there are two squares above and below the diagonal. The cells above the diagonal are selected to represent interfaces that have their source on the left side of a diagonal square to the top of a lower diagonal square. The cells below the diagonal are selected to represent interfaces that have their source on the right side of a diagonal square to the bottom of a higher diagonal square (Grady, 1994). Off-diagonal dark squares denote the transfer of information or activity dependencies.

Table 4.1:

A taxonomy of types of system element interactions (Pimmler and Eppinger, 1994).

Spatial	Associations of physical space and alignment; needs for adjacency of orientation between two elements.
Energy	Needs for energy transfer/exchange between two elements (e.g. power supply).
Information	Needs for data or signal exchange between two elements.
Material	Needs for material exchange between two elements.

The order of elements or activities can also be reshuffled with respect to the organization of the design elements. This is done as a means of reducing cross-element interfaces while also preserving functional allocation. This can be performed by relocating a row and its corresponding column in the matrix within a different subsystem based on the off-diagonal interface count. This will render the configuration unchanged, as the interfaces will automatically adapt to the new configuration. There are many analysis and reorganization tools and algorithms available that automate this procedure (Gebala and Eppinger, 1991).

The types of interactions that occur in DSM vary from one project to the next. Pimmler and Eppinger (1994) suggest a taxonomy for the types of system element interactions based on four main types of interactions: spatial, energy, information, and material, (as shown in table 4.1). These are similar to the type of functions discussed earlier in chapter two.

Table 4.2:

Scale used to represent different interactions (Pimmler and Eppinger, 1994).

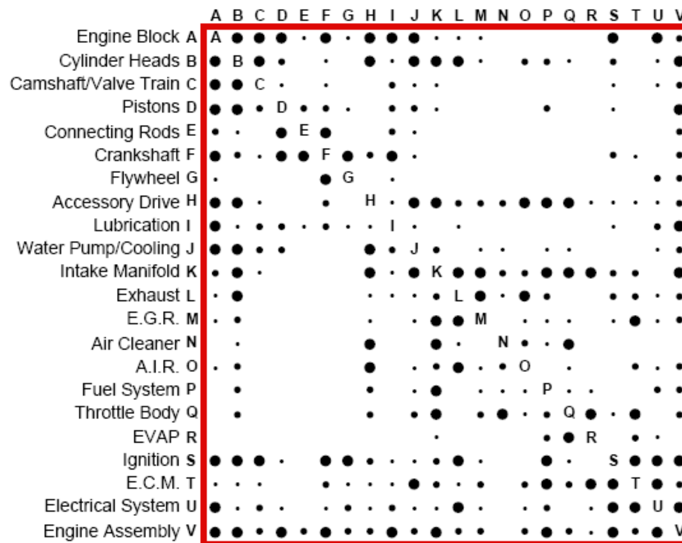
	Detrimental	Undesired	Indifferent	Desired	Required
Spatial	-2	-1	0	+1	+2
Energy	-2	-1	0	+1	+2
Information	-2	-1	0	+1	+2
Material	-2	-1	0	+1	+2

	Daily	Weekly	Monthly	None
Frequency of Interaction	● (large)	● (medium)	• (small)	

They also provide a quantification scheme for these interactions, where the square marks are replaced by numbers or colors. Table 4.2 illustrates different schemes to represent interactions. This provides a more comprehensive view of the overall system, as the relationships and interfaces between elements are investigated in more depth (figure 4.5).

Figure 4.5:

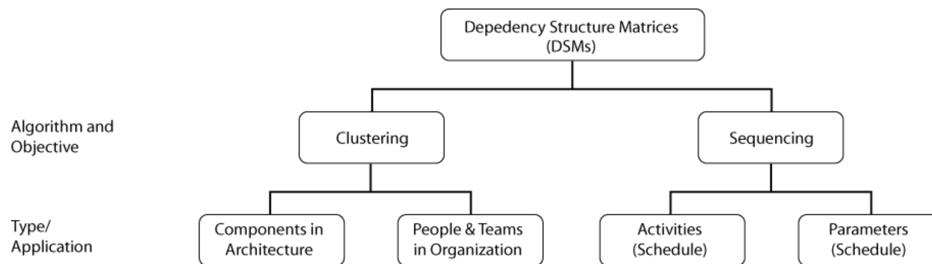
Application of a quantification scheme in a DSM (Pimmler and Eppinger, 1994).



Browning (1998) identifies four different types of DSM, their similarities, differences and applications (figure 4.6). These types are the component-based DSM, team-based DSM, activity-based DSM, and the parameter-based DSM. Activity-based DSM will be highlighted, as it pertains to the focus of this thesis, the informational and activity-based world rather than the physical world.

Figure 4.6:

Four different types of DSM (Browning,1998)



1- Component-Based or Architecture DSM: This DSM is useful for the modeling process of component relationships and enabling different decomposition strategies.

2- Team-Based or Organization DSM: This DSM is useful for the design of organizational structures that consider the information flow in design teams.

3- Activity-Based or Schedule DSM: This DSM illustrates in a highly visual format the modeling of design process iterations, input/output activity relationships, information flow structure, and project schedules in multi-activity systems based on activity information dependencies, sequences and arrangements. This capability is not provided by most conventional *PERT/CPM* techniques (Browning, 1998). Experience, historical data and the knowledge of design work to the most practical lowest level are all key players when it comes to building such a DSM model and prescribing activity sequence and project schedule. Designers can effectively control schedule risks by understanding which activities rely on and generate which types of information (Browning, 1998).

4- Parameter-Based or Low Level Schedule DSM: This DSM uses physical design parameter relationships for the purpose of planning design decisions and activities.

4.3 Iteration and coupling

The process of structuring the components of the MDDS design has an effect on the complexity of information flow within the MDDS and the time it takes to complete a design iteration.

Iteration is part of any design process. Its significance has been and can be defined as the repetition of activities to improve an evolving design (Eppinger et al., 1997). Smith and Eppinger (1996) explain that iterations occur for two reasons: the design fails to meet established criteria or new information is obtained since a prior iteration.

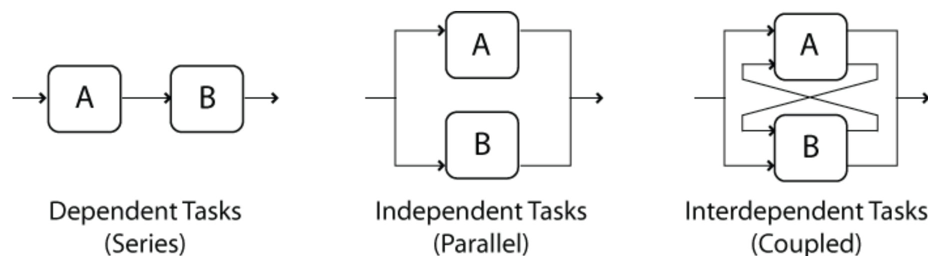
There are two types of iterations: intentional iterations and unintentional iterations. Iterations are intentional if the design is meant to progress to a specific desirable solution. Iterations are unintentional if new information that comes late in the process affects the design outcomes or results. This can result from fluid requirements, design goals and mistakes, or out of sequence activities (Browning, 1998).

In order to reduce the cycle variation and time of design development, unintentional iterations have to be minimized. This entails making sure that the sequencing of activities is enhanced, such that the correct information is available at the right place when it is time to perform the activity. It also means that the requirements are strongly defined as early as possible, the relevant constraints are provided and the mistakes are reduced to the minimum.

In addition to the tremendous magnitude of information involved in each iteration, the interdependency and coupling between design activities contributes significantly to the information flow complexity.

Figure 4.7:

Activity information flow and their equivalents (Eppinger, 1991).



Eppinger (1991) identifies three types of dependencies: serial (dependent), parallel (independent), and coupled (interdependent). Figure 4.7 illustrates directed graphs of three possible ways in which two design activities A and B can be related together.

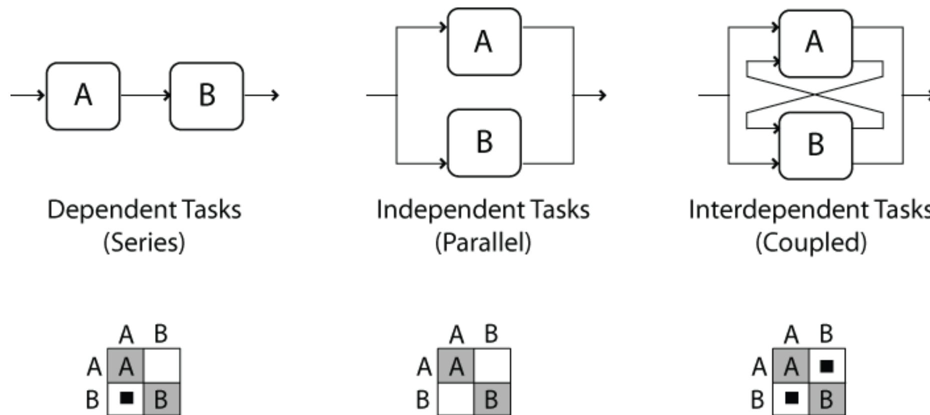
An activity is said to be dependent or performed in series if task A just requires the output of B (or vice versa). The two activities are said to be independent or performed in parallel if activities A and B can be done with no interaction between designers. The two activities are said to be interdependent or coupled if A requires information from B and at the same time B requires knowledge of the results of activity

A. Organizing and coordinating dependent or independent activities is much less challenging and time consuming than interdependent activities. This is due to the typical iterations of information transfer and design time required in the process (Suh et al., 1978).

Figure 4.8 illustrates two activity information flows and their corresponding DSM equivalents (Browning, 1998). Through this translation, system formulation models can be converted into activity-based DSM equivalents.

Figure 4.8:

Activity information flows and their corresponding DSM equivalents (Browning, 1998).



As shown in the previous figure, if the activities are inserted in the sequence in which they are to be performed, introducing a sub-diagonal mark in the activity-based DSM denotes information feedforward, while introducing a super-diagonal mark denotes information feedback. This indicates a counter-clockwise information flow. By resequencing the activities, that is by reshuffling the rows and columns of the DSM, a prescriptive DSM is revealed which reduces feedback in the process to the minimum. This minimization of feedback then gets the maximum possible interfaces below the diagonal of the DSM. The remaining feedback super-diagonal interfaces, apart from constraints, will be due to interdependent activities (Browning, 1998).

4.4 Process and Formulation Modeling

Designing the design process introduces technical challenges even for relatively well-structured problems. These challenges are the result of the large number of inputs that feed into the design activity, the huge magnitude of information that is created and transferred at various levels and stages, and the complexity of information flow within the process. The main goal of system formulation modeling is to capture the complexity of design processes and to work towards their improvement.

These challenges demand types of information representations that can aid the understanding of the design process and the structure of information flow. Formulation models and notations depend primarily on the idea that the design process as such has a similar underlying structure in spite of the fact that designs exist in different projects. A lot of effort has been made to introduce such representations and notations, starting from the early 1920's following the development of process charting theory (Graham, 2004). Other efforts involve process engineering and reengineering methodologies for supporting the analysis and documentation of design and organizational processes (Scholz-Reiter and Stickel, 1996).

In the following sections, we look at formulation models that implement certain notations. These notations are implemented in two different disciplines, namely system engineering notations that represent the physical artifacts, and software engineering notations that represent information activities.

4.4.1 Network Models

Network models use a variety of techniques to model design activities as networks of discrete-event activities or tasks with design information flowing in between. Some features can be added or deduced from theses network models such as cost, process time, sequence of data flow and transfer, etc. They are usually appropriate for the purpose of planning activities or tasks that are serial or parallel. Network models are influenced primarily by graph theory.

Next we will discuss data flow diagrams, functional flow block diagrams, and their variations.

4.4.1.1 Data Flow Diagrams

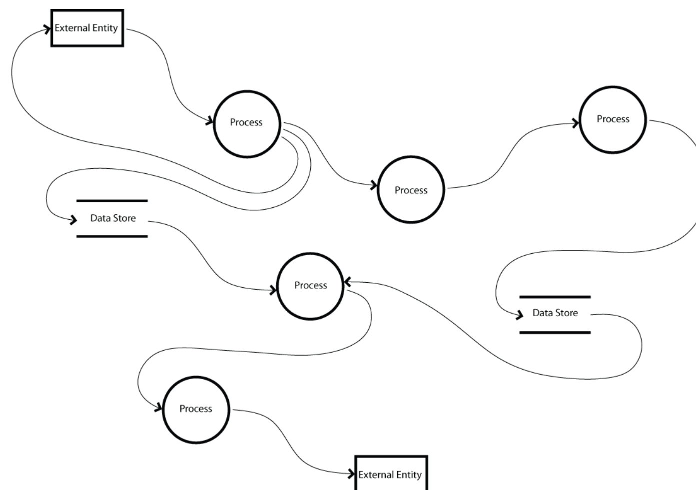
A Data Flow Diagram (DFD) is based on Directed Graphs. It stems from the software engineering discipline. A DFD is a graphical notation of the decomposition of a system. It basically highlights data flow between the different functions of a system (DeMarco, 1979). In such a process, a DFD identifies data transformations from input to output (Ward and Mellor, 1985).

DFDs focus on data flow rather than control. Therefore there are no control constructs that explicitly indicate the sequence of the processes. A DFD usually defines the content of each and every activity and the processes flowing in and out, but does not define the sequence in which these activities occur. Identifying time-ordering requires another technique.

There are various notations for drawing DFDs, but they usually consist of four main symbols: a process or activity, dataflow, a terminator, and a data store. Each node in the diagram represents a function or activity. Data triggers a process or activity, which in turn produces data. A link between two processes represents data flow between them. A terminator is a data source or sink located at the system boundary. A data store is similar but located within the system. The choice of process node location however is arbitrary in DFDs. This becomes crucial when there are a large number of nodes that could bring disorder to the model (figure 4.9).

Figure 4.9:

Data Flow Diagram (DFD).



4.4.1.2 Functional Flow Block Diagrams

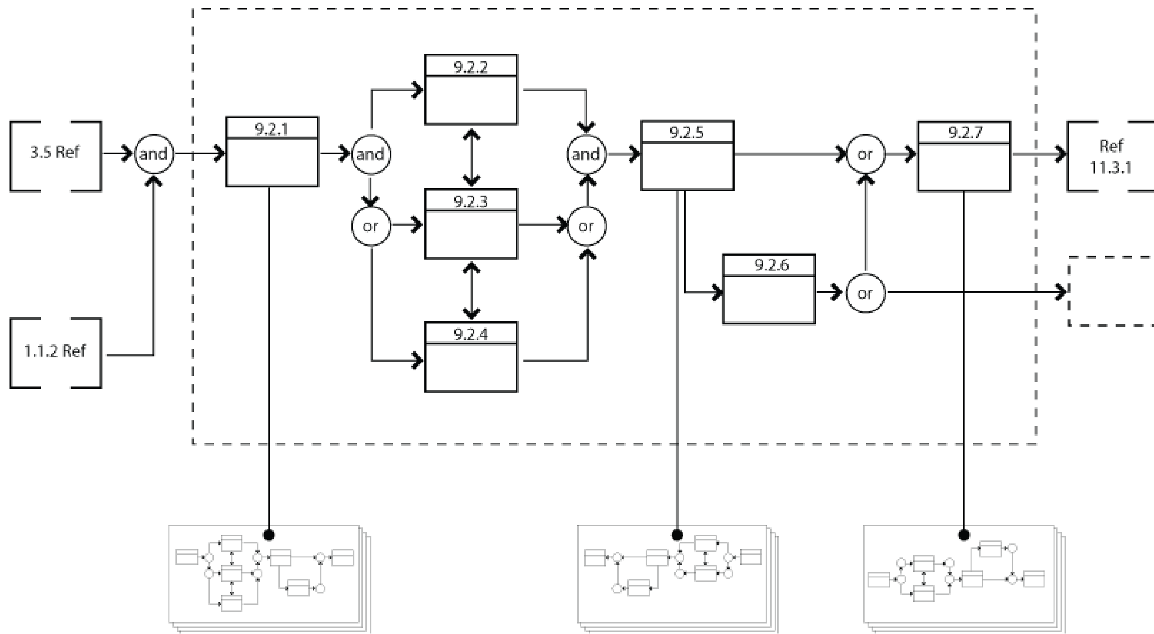
Function Flow Block Diagrams (FFBDs) were the first network model to be favored by system engineers and continue to be widely used today (Blanchard and Fabrycky, 1990). FFBDs define the decomposition of a specific system, and at the same time provide the logical and sequential relationship between processes, thus the time sequence of functional events can be illustrated.

Each function, represented by a block, occurs following the preceding function. Some functions may be performed in parallel or alternate paths may be taken. The diagram outlines the control of flow between processes and the order in which they are enabled and performed. The order can be specified from the set of available control constructs. Proper sequencing of activities and design relationships are established including critical design interfaces.

Figure 4.10:

Function Flow Block Diagrams (FFBDs).

Decomposition can be applied to define lower-level functions and sequencing relationships. This allows for vertical traceability through the levels and creates a hierarchical structure, which is a key step in developing the system architecture from which designs may be synthesized.



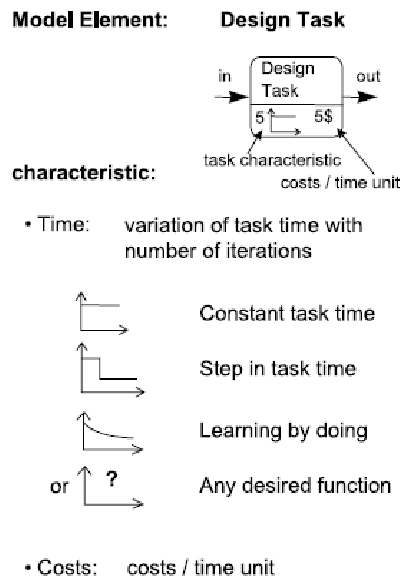
The basic FFBD diagram components consist of functional blocks, flow connectors, numbering, referencing, gates, and go and no-go paths. Functional blocks represent the system processes. They are connected by lines indicating functional flow. Arrows indicate the direction of this flow, which is usually from left to right. Numbers are used within the blocks to indicate the sequence of processes starting from the origin. As FFBDs are developed in a series of levels, a numbering scheme is used for each level. This traces functional flow between different levels. FFBDs can also contain references to other functional diagrams. Control constructs are used to direct the direction of flow. They are represented by gates of two main types: AND/OR. "AND" gates specify the need for parallel functions to satisfy requirements of all connected paths before proceeding. "OR" gates have the capability of providing passage if one of the connecting paths requirements is satisfied. Figure 4.10 shows the flow down structure of a set of FFBDs.

FFBDs provide an overall understanding of the system operation. They also point out locations where modification in procedure can possibly simplify this operation. What FFBDs do not provide however

is information about the type of data flowing across functions. Therefore it is a more function oriented rather than solution oriented approach (Long, 2002), where there is no specific answer to how a function is performed.

Figure 4.11:

Characteristics of the model element design review (Andersson et al., 1998).

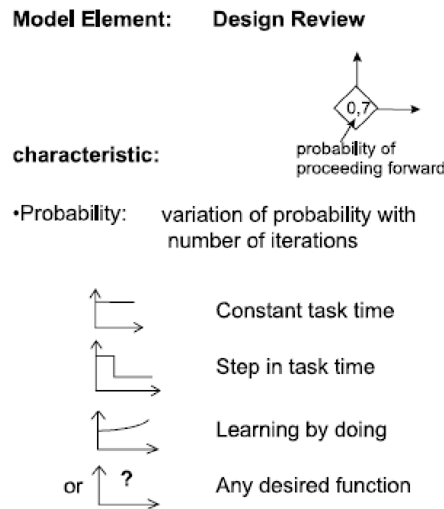


Over the years a number of variations and developments were made on FFBDs. These include Enhanced FFBDs, which provide additional representation of data as inputs and outputs to functions (Oliver, 1994). Further process characteristics were later added by Andersson et al. (1998), where design information flows were used to connect two main model elements: design tasks and design reviews.

Process characteristics, such as task cost per unit time (figures 4.11 & 4.12) and execution time, were introduced to design tasks. These can be adaptive to the advance of the design progress. The task characteristics were even made to vary with the number of iterations involved. For example, the first iteration in a design process, with considerable amount of CAD modeling, would require models to be created. In subsequent iterations, only modifications need to be made. This would result in more flexible and accurate model, and at the same time a less time-intensive process with step reduction in task time. Such tasks, where execution time changes with every design iteration, are modeled as "learning-by-doing" tasks with an associated learning curve function (Andersson et al., 1998).

Figure 4.12:

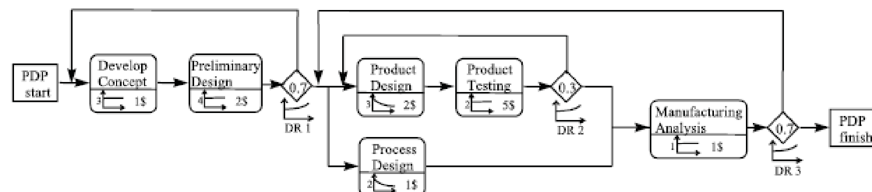
Characteristics of the model element design review (Andersson et al., 1998).



The design review model element illustrates the probability of advancing to subsequent design tasks (figure 4.13). Without this element the process would retreat back to earlier tasks (Andersson et al., 1998). The evaluation of this model element is done using a random function. Another relationship is established in design review model elements, where the characteristics of the design review are a function of the number of iterations. The relationship between number of iterations and design review probability can also be represented by a "learning-by-doing" function. The method is particularly useful for comparing design processes based on the global variables of process costs and lead time.

Figure 4.13:

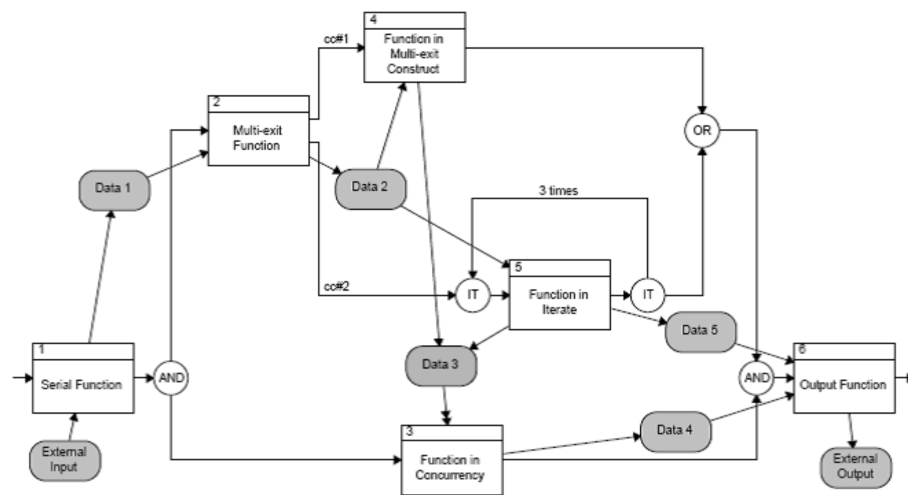
Design development process (Andersson et al., 1998).



In the field of software engineering IDEF0 was developed in order to represent data flow information. In software engineering models in general, functions are executed if there is both a data trigger and an enabling by control. A function is said to be triggered if the stimulus data becomes available to the function. A function is said to be enabled if the preceding function in the control flow specification is completely executed.

Figure 4.14:

Sample Enhanced FFBD (Long, 2002).



4.4.2 Formulation Modeling Languages

4.4.2.1 Unified Modeling Language

UML (or Unified Modeling Language) is a general-purpose, standardized visual specification modeling language. It uses graphical diagrammatic representation to create an abstract model of a system, enabling software developers to model computer applications. This model is referred to as a UML model. It is mostly used to model structure, behavior, and architecture, even business process and data structure. It has introduced a revolution in the flexibility of reading and circulating system structure and design plans.

The release of UML open standard by the Object Management Group (OMG) in 1997 involved the joint efforts of modeling languages of three main system development methods: Grady Booch's Booch '93 method, James Rumbaugh's Object Modeling Technique (OMT)-2 method, and Ivar Jacobson's Object Oriented Software Engineering (OOSE) method. Together with methods from information systems and engineering practices, OMG formed a new modeling language. Concepts from many object-oriented methods were also integrated with UML aiming at object-oriented support.

One of the main reasons UML is used as a standard modeling language is that it is a language, as opposed to a methodology, so it easily fits into any way of doing business without much modification. As it is not a methodology, it does not require any formal work

products. It provides, however, many diagram types that help in the understanding of an application under development when used within a given methodology.

In addition, UML is programming-language independent and platform-independent. Its tools are used at length in J2EE and .NET shops. It has thus enabled software developers to focus more on design and architecture due to this stable and common design language. Due to the broad and rich coverage emphasized in the real-time systems domain, UML is used in many engineering problems, such as single process, single user applications as well as concurrent, distributed systems.

UML models are different from the represented set of diagrams of a system. A diagram is a partial graphical representation of the model. The model at the same time contains written use cases which act as documentation that drives the model elements and diagrams.

There are three main processes involved in UML models: visualizing, constructing, and documenting. Visualizing involves using diagrams for communicating the model as an idea into an expression in the form of diagrams. Constructing uses these visual illustrations in a prescriptive manner to build the system. Documenting involves using models and diagrams to capture knowledge of the requirements and system throughout the process.

UML Views

UML defines thirteen types of diagrams which represent three different views of a system model. Six diagram types represent static application structure; three represent general types of behavior; and four represent different aspects of interaction. In the following sections I will demonstrate some of these views and diagrams.

Static structural view

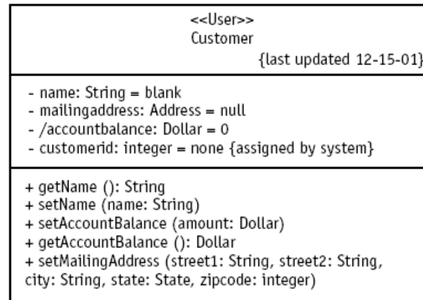
This view emphasizes the static structure of a system, meaning what must exist in the modeled system. This is done by using objects, attributes, operations, and relationships. Structure diagrams include the following diagrams: class diagrams, object diagrams, component diagrams, composite structure diagrams, package diagrams, and deployment diagrams.

A class diagram visually represents the classes, or entities, of an application and the relationships between them. It depicts the overall static structures of the system. The notation of a class in a

class diagram is a rectangle with three horizontal sections (figure 4.15). The upper section represents the class name, the middle section consists of the class attributes, and the lower section represents the class operations and methods.

Figure 4.15:

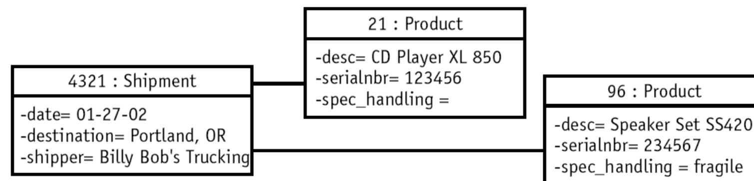
Sample class object in a class diagram (Pender , 2002).



In general, a class diagram contains the following types of elements: a class representing a general concept an association representing a relationship between classes, an attribute representing the knowledge of objects in the class, and an operation, representing what objects in the class can perform as operations. Association relationships are represented as solid lines if both classes are aware of each other and lines with open arrowheads if the association is known by only one of the classes. Inheritance relationships, on the other hand, are drawn as lines with arrowheads pointing to the super class.

Figure 4.16:

A complete class diagram (Pender , 2002).



A component diagram (figure 4.16) shows the implementation of a system. It provides a physical view of the system, depicting the dependencies that the software has on other software components in the system. In general, component diagrams consist of major system components and their relationships. Component diagrams have the following types of elements: a component representing a part of the system that exists while the system is executing; and a dependency relationship, which represents that the client component consumes or depends on the supplier component. Similar to object-oriented methods, component-based diagrams are based on principles of abstraction, encapsulation, generalization, and polymorphism. The main difference however lies in focusing on components rather than objects.

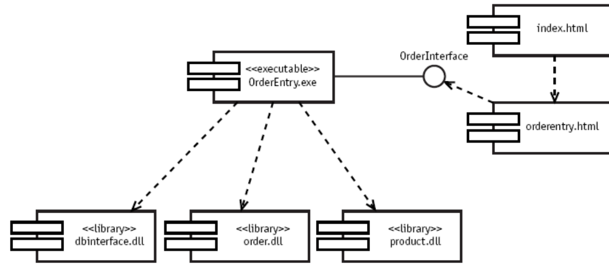


Figure 4.17:

A component diagram shows interdependencies of various software components the system comprises (Pender , 2002).

A deployment diagram (figure 4.17) illustrates the implementation environment of a system. In this sense, both component and deployment diagrams are specific types of what is known as implementation diagrams. The deployment diagram describes the physical deployment of a system in the hardware environment. It illustrates where different components of the system run physically and how they communicate together, in addition to modeling the physical runtime of the system.

The notation system in a deployment diagram is analogous to that used in a component diagram. The concept of a node is added however. A node here represents either a physical or a virtual machine node. A component that resides on a node is nested inside the node. Deployment diagrams have the following types of elements: a node representing a resource that is available during execution time; and a communication association, which represents a communication path between the nodes.

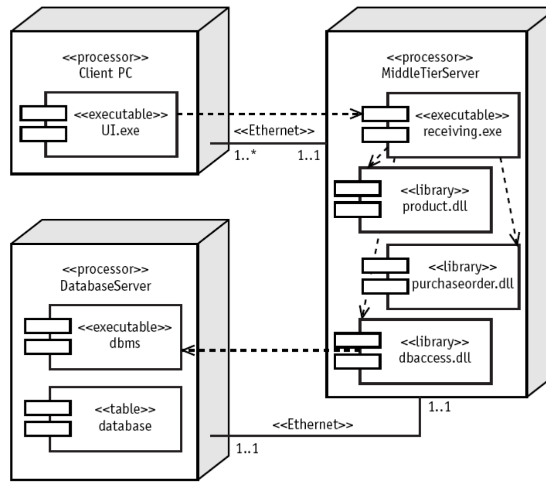
Behavior view

This view focuses on the dynamic behavior within a system, including changes to the internal states of objects. It also stresses on the collaborative activities and decisions among objects, describing what must happen in the modeled system. Behavior diagrams are primarily flowcharts and DFDs that are used to acquire the general flow of the code. They include the following diagrams: use case diagrams, activity diagrams, and state machine diagrams.

A use case diagram (figure 4.19) basically works on communicating high-level functions of the system scope. In doing this, it captures the functional requirements of a system, thus helping development teams visualize those requirements. Use case diagrams are thus widely used by software engineers. The diagram is useful in the process of describing those functional requirements during the analysis, design, implementation and documentation stages.

Figure 4.18:

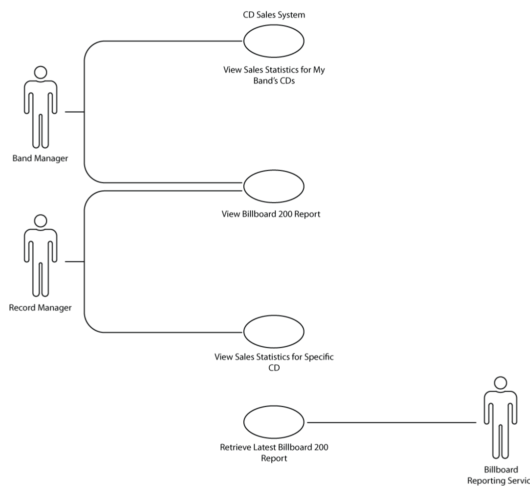
Deployment diagram (Pender , 2002).



The use case diagram clearly shows the relationship of actors, who represent human beings interacting with the system, to basic processes, in addition to the relationships among different use cases. It therefore does not provide all the functions required in interface management or in defining scenarios, as it relies basically on demonstrating human initiated functionality. Typically, a use case diagram shows groups of use cases. This is done by either showing the complete set of use cases for the whole system, or by showing a functionally related group of use cases

Figure 4.19:

Sample use-case diagram.



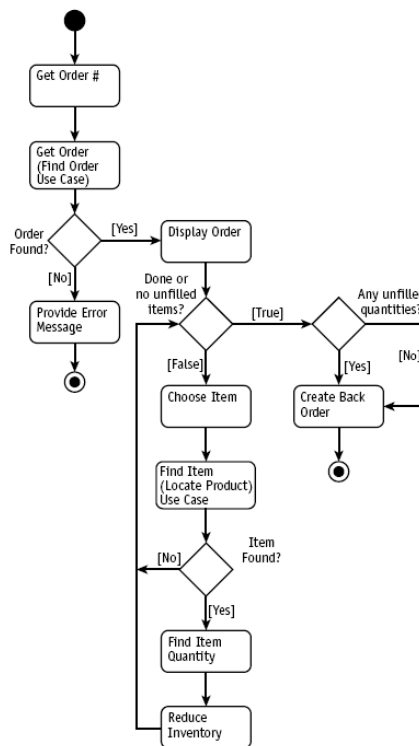
An activity diagram depicts the procedural flow of control between two or more class objects while processing an activity. An activity diagram can model both high-level business processes at the business unit level and low-level internal class actions. An activity is

essentially modeled by drawing a rectangle with rounded edges. This rectangle encloses the activity name. The notation system is similar to the state diagram. Activities can either be linked to other activities through transition lines, or to decision points. These points then link to the various activities controlled by the state of the decision point. At the point of termination of the modeling process, an activity is connected to a termination point. Activities can optionally be grouped into "swimlanes" (figure 4.20). These are used to denote the object that in reality performs the activity.

State diagrams, which are also known as statechart diagrams, show the lifecycle of a system component. State diagrams model the different states or conditions which a class can exist in. More importantly, they model the process of class transitioning from one state to another. Usually every class possesses a state, but should not necessarily have a state diagram. Along system activity, only those classes that have three or more potential states are considered interesting to model.

Figure 4.20:

Activity diagram with 3 swimlanes (Pender,2002).



State diagrams have the following types of elements: a state which represents the condition of a component; an event describing the occurrence of message receipt; a transition; an initial state; and a final state. As a component is created, it enters an initial state. The

transition starting from the initial state is labeled with the event that creates the component. As the component enters its final state, it is destroyed. The transition to the final state is labeled with the event that destroys the component (figure 4.21).

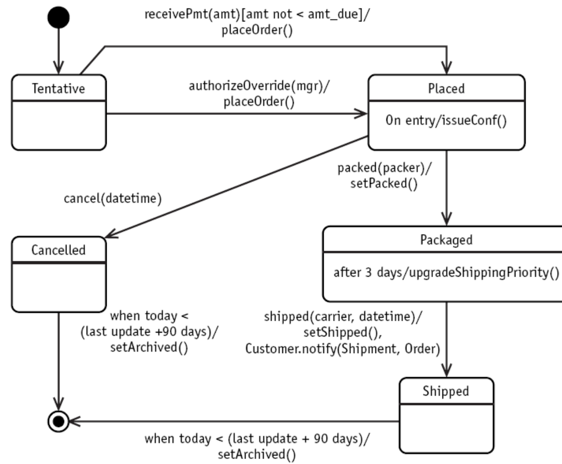


Figure 4.21:

Statechart diagram showing the various states that classes pass through in a functioning system (Pender, 2002)

Interactions View

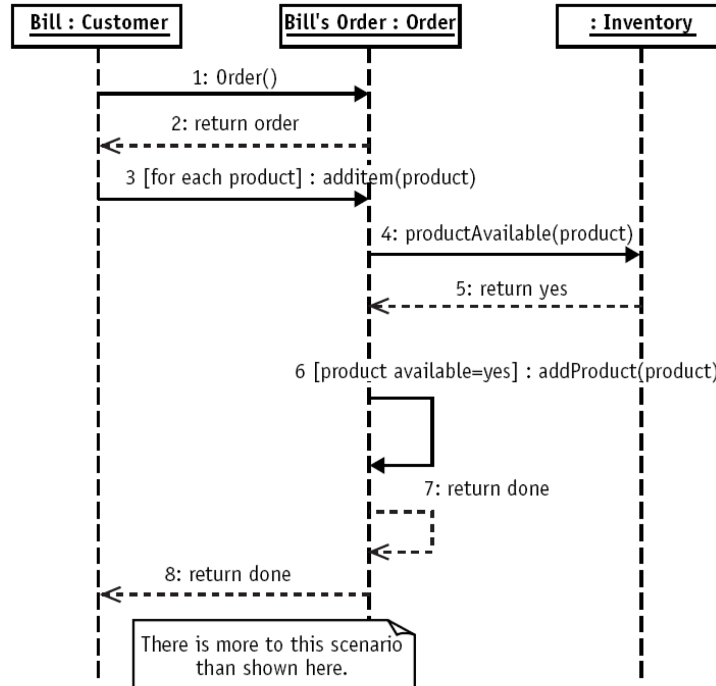
The Interaction diagram is a subset of behavior diagrams. It focuses however on the flow of data and control among the objects in the modeled system. Interaction diagrams include the following diagrams: sequence diagrams, communication diagrams, timing diagrams, and interaction overview diagrams.

Sequence diagrams (figure 4.22) are primarily interested in the time and ordering factor. Widely used among software engineers, sequence diagrams show how different physical components interact over time. Interaction in this context refers to the exchange of messages or calls. Calls among objects as well as different calls to different objects can be depicted, all visualized according to time sequence. The content of sequence diagrams is concerned with specifying the data flow between a subset of system components. They can illustrate a detailed flow for a specific use case or a segment of a particular use case.

A sequence diagram has two basic dimensions: the vertical dimension, which shows the time sequence of messages; and the horizontal dimension, which shows the objects involved in the interaction, specifically the instances to which the messages are sent. Unlike FFBDs, EFFBDs and behavior diagrams, however, sequence diagrams cannot characterize control in terms of constructs. Specification of control in the sequence diagram notation is incomplete and consequently cannot be implemented.

Figure 4.22:

A sample sequence diagram (Pender,2002).



In general, there is no restriction as to the appearance of all UML components on any types of UML diagrams. In terms of notation, usually the presence of a comment or note is allowed in a UML diagram, so that intent, usage, or constraints can be expressed and explained clearly. This is traced back to the conventional notation system used in engineering drawings.

4.4.2.2 Systems Modeling Language (SysML)

OMG SysML™ is a general-purpose graphical modeling language characterized by having computer-sensible semantics (OMG, 2007a). The main purpose of this language is the identification, analysis, design, and verification of complex systems. In a way, SysML adapts UML™, which is primarily used for modeling software-intensive systems, for the purpose of systems engineering applications. Similar to the UML approach in unifying modeling languages in the software industry, SysML reuses a subset of UML 2 to unify the wide range of modeling languages, tools and techniques currently in use by systems engineers.

The history of SysML goes back to 2001 when the International Council on Systems Engineering's (INCOSE) Model Driven Systems Design workgroup decided to customize UML for systems engineering applications. Two main bodies, the INCOSE and the

Object Management Group (OMG) (which maintains UML specification), collaborated as a result and jointly developed with the assistance of other groups the specifications for the SysML in March 2003 (OMG, 2007a).

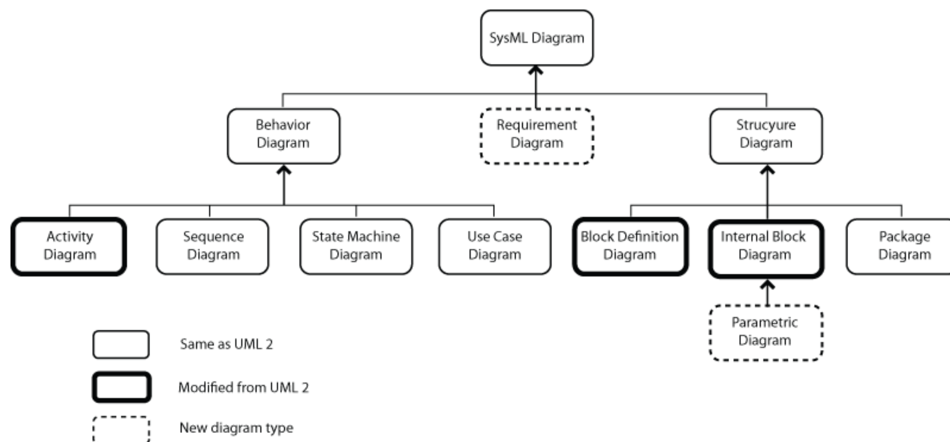
SysML uses UML 2.0 and its extensions as its basic foundation. Therefore both systems engineers using SysML and software engineers who model using UML 2 can collaborate effortlessly on models of software-intensive systems. This enhanced communication among participants in the systems development process advances interoperability among modeling tools. It is most likely that SysML will be customized to model domain-specific applications, such as automotive, aerospace, communications, and information systems.

Figure 4.23 illustrates the SysML diagram taxonomy, representing the concrete notation for the diagrams, together with the corresponding specification of the UML extensions. Compared to UML, SysML is a smaller language, both in diagram types and total constructs, as it reduces many of UML's software-centric constructs. This makes it an easier language to learn and apply, and much more flexible and expressive.

SysML, like UML, supports allocation tables, a tabular format that is dynamically derived from allocation relationships. While UML provides only limited support for tabular notations, SysML is characterized by flexible allocation tables that support requirement, functional and structural allocation. SysML constructs for model management extend UML capabilities and support models, views, and viewpoints.

Figure 4.23:

SysML diagram taxonomy (OMG,2007b)



SysML implements a total of nine diagram types, seven of which belong to the original thirteen UML 2.0 diagrams. It adds two other diagram types: requirements, used for requirements management; and parametric diagrams, used for performance and quantitative analysis.

A requirement specifies a condition that should be met. A requirement may specify a function that a system must execute or a performance specification a system must achieve. The requirements diagram can depict the requirements in graphical, tabular, or tree structure format. Other diagrams can also have requirements appear on them to show their relationship to other modeling elements. Modeling constructs are supplied in SysML to represent text-based requirements and their relation to other modeling elements. The requirements modeling constructs were developed to bridge between traditional requirements management tools and the other SysML models (OMG, 2007b).

Parametrics primarily supports engineering analysis of critical system parameters, well known as a crucial aspect of systems engineering. This analysis includes the evaluation of performance, reliability, and physical characteristics (OMG, 2007b). Parametrics addresses the gap in previous modeling languages such as UML, IDEF, and behavior diagrams. It also provides a mechanism that deals with problems in non-standardized engineering analysis models. Previous non-standardized engineering analysis models lack the integration and synchronization with system architectural models, which specify the behavioral and structural aspects of a system, due to the complexity and diversity of engineering analysis models. Parametrics integrates engineering analysis models with system requirements and design models for behavior and structure. It is also represents constraints in order to capture other types of knowledge beyond engineering analysis.

With these augmentations, SysML can model many systems, including hardware, software, information, processes, personnel, and facilities. Principal terminology in SysML parametrics includes the following terms:

- *Constraints* are similar to equations. This is useful in most engineering problems. A *constraint block* defines this equation in a way that makes it reusable. A *constraint property* is a specific instance of usage of a generic constraint block (for example, supporting engineering analysis of a particular design).
- *Parameters* represent variables of an equation or a

constraint.

- *Value properties* represent any measurable attributes of a system architectural model or its components that are subject to analysis (e.g. mass). Through binding, the generic equations are linked to the value properties that specify the system and its components. Thus, the value properties are said to be bound to the parameters of a constraint.

5. Modeling

5.1 What is a Model?

In general, a *model* is an imitation or approximate representation of a system or of complex functions (Papalambros and Wilde, 2000). It is a simplified or abstract view of the complex reality using a physical, mathematical, or logical representation of the system of entities, phenomena, or processes. It may focus on specific views, thus facilitating the understanding and analysis of complex problems through decomposition.

Modeling, as an efficient communication tool, illustrates how systems and processes work and induces creative thinking about their enhancement. It is used by artists, architects, engineers, designers, planners, operations researchers, economists, managers, scientists, and others to study, plan, design, or control systems and artifacts. The basic concept that these uses build upon is the fact that model behavior represents to a great extent that of the system or artifact in an abstract and simple manner.

Modeling usually reduces cost, risk, and flow time of certain tasks. Most often modeling remains the sole method to perform tests and experiments as it is sometimes unfeasible, costly or disruptive to use the actual system or artifact for these purposes (Averill, 2006). The system or artifact that is to be modeled might not even exist in reality but we may want to propose multiple configurations and study different alternatives to find out how it should be initially designed. It is therefore important to construct a model and study it as a substitute for the actual system (Averill, 2006).

Models represent a simplification because they extract only the highly significant aspects of the real artifact for efficiency, reliability, and ease of analysis purposes. Models can replace a specific phenomenon in an unknown field with representation in another field with which the user is more familiar. Phenomena can thus be made simple, many relevant attributes can be extracted, and effects can be scaled in space and time to acquire tailored levels of detail while maintaining the modeling experimentation convenience. The

question always remains as to whether the model reflects accurately the system or artifact for the purposes of the decisions that are to be made.

Models usually fall into one of two categories, physical or symbolic models (Jacoby and Kowalik, 1980). A model is considered a physical or material model if the system representation is a tangible and material representation (Papalambros and Wilde, 2000), comprising model elements made of materials and hardware. These models typically include smaller scale versions of real objects, such as a ship model, and are basically intended for experimentation, study and display (Maki and Thompson, 2006).

A model is considered to be a symbolic or formal model if the system representation is theoretical or symbolic, and conducted by tools developed primarily for abstraction purposes, such as drawings, logic, mathematics or verbalization (Papalambros and Wilde, 2000). A building blueprint is considered a pictorial symbolic model. Words connected with logical statements form complex verbal symbolic models. Computer languages are an extension of these ideas as well (Papalambros and Wilde, 2000). Here we are concerned with symbolic models and specifically mathematical models. These are models that can be implemented in a computer environment.

5.2 The Mathematical Model

A mathematical model is a formal model that comprises symbols, assumptions about the symbols, the relations among the symbols, and connections between the actual model and these symbols and relations (Maki and Thompson, 2006). Within a design problem it consists of a set of quantitative and logical statements that represent relevant features of a specific artifact or system in terms of mathematical concepts, symbols and language including variables, parameters, and relationships such as equations and inequalities (Jacoby and Kowalik, 1980; Maki and Thompson, 2006).

A mathematical model becomes a computational model as soon as its associated equations are coded into a computer program where it can be studied numerically and graphically (Maki and Thompson, 2006). Simulation, as one of the applications that involves intense computation, deals primarily with the process of designing a model of a system and conducting experiments on that model. The relationships in the model are manipulated to observe how the model reacts, and how the system would eventually react accordingly if the mathematical model were valid (Averill, 2006). This allows for testing hypotheses at a much lower cost than actually

performing that activity in reality.

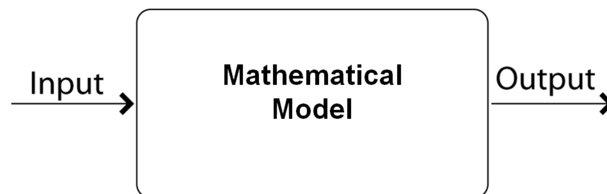
If we assume the output of the system is y, the input is x, and f is a mathematical function that calculates a value of y for each value of x. Then the following equation is a mathematical model of the system:

$$y=f(x)$$

This representation is merely symbolic. The actual function can be a system of algebraic or differential equations or a computer-based subroutine. In order to provide a mathematical model of a design, it has to be fully defined through assigning values to every single involved quantity. These values have to satisfy the mathematical relations that represent the performance of a specific task (Papalambros and Wilde, 2000).

Figure 5.1:

Block diagram representation of a mathematical model.



We also can represent this mathematical model of the system using a block diagram representation (figure 5.1). If a set of relations representing the physical system mechanism were enclosed in a block or box that could be only approached through input and output terminals, that box is known as a closed box (Jacoby and Kowalik, 1980). It is impossible to tell apart from other boxes that can generate the same outputs from the same inputs. By adding input or input terminals, these boxes can be made more "open", that is to say they can be made distinct from each other. The only difference then between an open and a closed box is a quantitative and not a qualitative difference, as the number of added terminals affects the closed and open nature of that box.

5.2.1 Elements of mathematical models

Mathematical models generally consist of a model interior, a boundary, and a group of boundary and initial conditions that represent aspects of the artifact environment relevant to the modeling experiments.

The boundary and the set of boundary and initial conditions represent aspects of the system and artifact environment that relate to the modeling experiments. The model interior on the other hand is mostly structured and comprises interconnected components that represent parts of the artifact or system. Overall, the model interior includes variables, parameters, constants, mathematical relations, and algorithms.

Variables represent different system states by assuming different values that are possible within an acceptable range. The solution we usually seek for the model experiment consists of the values of the variables satisfying the modeling expressions. The choice of variable types depends on the system we are dealing with.

Parameters represent fixed quantities that are assigned one particular value in a particular model experiment but may nevertheless be changed from one experiment to another. They are assigned and fixed by the model application and not the underlying phenomenon.

It is important to distinguish especially in the modeling stage between variables and parameters. Variables that are fixed in a particular modeling experiment are considered parameters. Selecting which quantities will be assigned or categorized as variables or parameters is a subjective decision. It is basically determined by selections in hierarchical level, boundary isolation, and intended use of the model (Papalambros and Wilde, 2000).

Constants are quantities that are fixed and assigned by the specific phenomenon and not by the model statement. They are usually natural or design requirement quantities that cannot be affected or changed by the designer.

Mathematical relations, such as equations or inequalities, merge the variables, parameters and constants together. The most difficult part of modeling is the issue of stating these relations which aim to describe the system function within the conditions and constraints set by its environment (Papalambros and Wilde, 2000).

Algorithms are broadly defined as step-by-step procedure for solving a problem or achieving some end (Daffa', 1977). The words *algorism* and *algorithm* stem from Algoritmi, the Latinization of the name of Al-Khwarizmi, a Muslim mathematician born around 780. The name was later modified from the Arabic name to Medieval Latin *algorismus*, then to Old French & Medieval Latin, then to the Middle

English word *algorisme*, which was later altered to *algorithm*.

Algorithms may be expressed differently according to the medium where they are tackled. Algorithms can be computed by either humans or machines, so they are not machine-dependent. An algorithm can take the form of numbers, verbs, actions and drawings in the case of humans, while it takes the form of numbers in computers.

Finding an algorithm that can solve a given problem effectively has long been difficult for mathematicians and eventually led to the theory of computability by Turing in 1936 (Kalay, 2004). This difficulty proved not only a matter of which problems lend themselves to mechanical or computational solutions, but also of finding the most effective algorithm to solve these problems. The algorithm must accomplish its designated task within a sensible time frame and using reasonable computing resources.

Some problems are appropriate for either iterative or recursive implementations. The process of iteration in general can describe the repetition of any process within a computer program. It can also describe a particular form of repetition with a variable state. Repetitive constructs are employed in these iterative algorithms, such as loops and often some additional data structures such as stacks, in order to solve the given problems.

Recursive algorithms, a method common to functional programming, describe those algorithms that call or refer to themselves continuously and repeatedly until a specific condition is met. The strength in using recursion lies in the ability to define an infinite set of objects by a finite statement and an infinite number of computations by a finite recursive program even if it does not overtly contain any repetitions (Wirth, 1976).

In computers, algorithms are usually executed in three ways: serial, parallel or distributed. Serial computing is where algorithms are computed by a processor only one at time sequentially. In parallel computing, algorithms are computed simultaneously by more processes, thus requiring the algorithm structure to enable such computing. The third type of algorithms, distributed algorithms, can use several computers connected through networks.

5.2.2 Constructing Mathematical Models

This section is concerned with the scope of the modeling process which generally relies on the project objectives, performance

measures, availability of data, credibility concerns, computer constraints, time and money constraints, and the time frame for the study and the required resources. The scope of the model is also highly dependent on and driven by design intent which is the group of objectives that a certain system or artifact expresses or helps realize. This is categorized into two main classes: quantifiable design intents and qualitative design intents. Quantifiable design intents can be identified, expressed and managed easily as their performances can be measured numerically, whereas qualitative design intents cannot. It follows that defining the scope is extremely reliant on the model developer and designer.

Many experts are required for successful model construction. It is important to be able to identify those aspects of the design problem that are relevant to answering questions about the artifact or system behavior in a real world situation. Usually this is done by individuals who are knowledgeable and familiar with the origin of the design problem and the involved discipline. These individuals and model developers are the ones who initiate the first steps in model building. They do this by closely studying the system and its conditions in detail and recognizing analogies to other conditions before moving on to basic assumptions about the problem under study. The intent is to define the boundaries of the models, which is an important challenge in model construction. It involves primarily selecting the elements to include, the features and attributes to consider, and the level of detail.

Selecting the level of detail of a model is not an easy task; it is rather an art. The ultimate limit of the model detail tends to mimic exactly the artifact or even become the artifact itself. This limit can be achieved in very few design situations (Jacoby and Kowalik, 1980). One good practice is to start with a relatively simple bounded scope that comprises the most basic elements of the system and then add more layers of detail or new elements according to the accumulated understanding developed along the process. This way models start as rough estimates of reality and then progressively reach the complexity of the studied system or artifact.

The intent underlying the mathematical model is usually guided by the intent that it represents and the purpose it serves. This is mainly a challenge as it is an economic decision which is based on model developer experience with no magical recipe. There is often no need to model each and every aspect of the system in order to make effective decisions, as this could result in high model execution time, or conceal valuable system or artifact factors. Level of detail is not usually analogous to higher-fidelity models, as some models are

better broken to certain combinations to guarantee execution speed and a degree of specificity. The aim is to make the design problem as simple and accurate as could be through a process of simplifications and approximations. This is done by eliminating information that is not necessary and simplifying the necessary information as much as possible, and thus not all the detail of the real world has to be taken into account and fewer objects and processes can be dealt with (Maki and Thompson, 2006).

Another key step is identifying the operative processes at work with the goal of expressing them symbolically. This process often demands high levels of creativity where all the quantities and processes are represented through mathematical operations and symbols. At the same time, it determines the validity of results, as the unsuitable mapping between the real and mathematical worlds could lead to results that are not that useful. It is therefore important to observe that the mathematical operations are error-free. This is done by observing that there has been no significant omission in the step from the real world to the mathematical model and that the mathematical model reflects all the necessary aspects of the real model. Ideally everything observed should be accounted for in the mathematical conclusions, and all the predictions would thus be verified by experiment. This does not typically occur, however, especially if it is the first attempt. The usual situation is that some of these conclusions agree with the experiment outcomes and some do not. Every step in the process has to be scrutinized again in this case (Maki and Thompson, 2006).

Usually the whole process operates iteratively through continuous refinements until an acceptable model is generated. The outcome is not necessarily a unique mathematical model, but it is shown that some of the several generated models can be distinctly better than the others. It may also occur that a number of models turn out to be useful for the same situation, where each model contributes to one but not all aspects of the problem under study. There is therefore not necessarily a “best” model. Choosing the model to use relies basically on the exact questions of the study (Maki and Thompson, 2006).

A model is generally judged by its performance in the tasks it was originally intended for, whether the model was designed to explain, predict or facilitate decision-making. In the case of explanation, the model is judged by its ability to offer a suitable description of the observed phenomena. In the case of prediction, it is judged through the degree of precision of the predictions it was based on. In the case of decision-making, it is judged by the efficiency and precision

of decisions it was based on, in comparison to decisions that are based on other criteria (Maki and Thompson, 2006).

Once the model and a mathematical structure are defined programming and computer software can be used in order to simulate the situation under study. Graphical and numerical output can be generated through simulation over different sets of conditions or relations of interest. That output is then evaluated, and the results are utilized to conclude about the situation under study or make a specific decision. The primary concern of this evaluation, however, is how representative the output is.

5.2.3 Types of Mathematical Models in Design

Coyne *et al.* 1990 have written that "In modeling design we do not attempt to say what design is or how human designers do what they do, but rather provide models by which we can explain and perhaps even replicate certain aspect of design behavior."

To model a design mathematically we must be able to define it fully. Designers and engineers regularly utilize mathematical models to perform typical design activities. These activities include generating one or more physical configurations, known as synthesis. They also include studying the performance and behavior of these configurations through engineering and science which is known as analysis. Designers and engineers then have to make design decisions about the results, which is known as evaluation. Finally they have to devise mechanisms for searching for the best alternative(s), which is known as optimization (Papalambros and Wilde, 2000). With a powerful tool like modeling, complex systems can be synthesized, analyzed, evaluated and optimized. Mathematical models are especially well-suited for design due to their flexibility and ease of modification (Jacoby and Kowalik, 1980).

Synthesis Models

Configuration synthesis, a unique and open-ended attribute of the design process, is well known to be the most significant and innovative part in design evolution. The synthesis process comprises decisions involving the overall arrangement of parts, how they can be assembled together, in addition to geometrical forms, kinds of motion, force transmission, etc (Papalambros and Wilde, 2000). It also enables the fulfillment of requirements by generating physical and informational structures, including machines, software, and organizations (Suh, 1990).

Although this process has historically been one that required human ingenuity and skill, many researchers have attempted to structurally formalize it, aiming at achieving shorter design cycles and more robust solutions. With the introduction of computers, formal design methodologies and structured algorithmic descriptions, automatic design synthesis can be accomplished computationally. This computational approach has the advantage of managing and tackling problems that are not open to solution by humans.

Analysis Models

Analysis models are developed according to principles of engineering science (Papalambros and Wilde, 2000). These models, which incorporate different analysis results, are constructed with the purpose of predicting the overall performance of the design. In the analysis process, engineers usually construct a descriptive mathematical model. This model constitutes a hypothesis and estimate of how the artifact could possibly work, or how unpredicted events could affect that system or artifact.

Evaluation Models

Evaluation models aid the process of selecting good designs that constitute a compromise of several different requirements. This means that a design can be altered to create different alternatives with the ultimate goal being to choose the most desirable alternative. A decision has to be made once there is more than one alternative to choose from.

The model helps provide a clear explanation, prediction and a foundation for objective decision-making. The rational selection of an alternative requires a criterion which helps evaluate all alternatives and rank them according to best fit. The criterion used in such models is known as the objective of the model (Papalambros and Wilde, 2000). It is not unique, however, and its selection will be affected by a variety of factors. These include the design application, timing, point of view, the designers' own judgment, and the position of the individual in the hierarchy of the organization (Papalambros and Wilde, 2000).

Optimization Models

This type of mathematical model enables moving from one configuration to the other in the ongoing search for better solutions, but more importantly it is established with the aim of control and guidance.

Optimization techniques are often used to determine potential design configurations by optimizing them according to the functional objectives and requirements developed in evaluation models. The solution to the problem is generally developed through solving the mathematical model which consists of an objective function that is to be optimized, and a group of constraints that act as resource limitations (Bahrami and Dagli, 1994). Optimization usually offers crucial solutions in situations where design problems can be formulated according to the objective and functional requirements. In this optimization process, simulation can be used in computing variables of the design vector. If not appropriate, variables of the design can be altered and the process is then repeated.

5.3 Synthesis Models

5.3.1 What is a Synthesis Model?

Historically synthesis has been a human effort that usually required skill and creativity. Through several attempts to shorten design cycles and achieve more robust solutions, design synthesis processes have been formalized. In the computer age, formal design methodologies, together with algorithmic descriptions, could be used to obtain automatic design synthesis computationally and handle problems that that may exceed human capabilities.

According to specific user needs and technical limitations, analysis and synthesis models and algorithms have evolved fully independently of each other although they are very closely related. Both fields are crucial to the design of complex systems, as design solutions are only identified through synthesis mechanisms, while there is no scientific basis for any of those solutions without analysis.

Synthesis models gain strength from being able to generate solutions that may be unpredictable or surprising, even to the designers themselves. These unpredictable solutions may be either beneficial or not. The interesting notion, however, is that synthesis models exhibit inherently a theoretical interest as they challenge the basic principles of the designer-machine relation. This can lead to results that are mostly outstanding and better than intended.

Synthesis models are primarily based on the fundamental concept of computation, where input information is operated on by functions in a computer which follows some algorithms to generate some sort of output. These models rely on analyzing specific design processes and programming them into computers. By developing shape manipulation algorithms, designers can use computers to generate many shape configurations. Designers use these systems to explore formal design concepts, describe the generated forms via computer algorithms, and ultimately use the algorithms as design tools for developing product forms.

In general, generative synthesis algorithms present a powerful formalism that can generate solutions within the design space defined by the system design language. By using these generative synthesis algorithms, computers become powerful design assistants and go beyond their traditional roles in design such as drafting, visualization or analysis. By embedding various performance requirements from the different design disciplines in the synthesis algorithm, many multidisciplinary alternatives can be created, thus

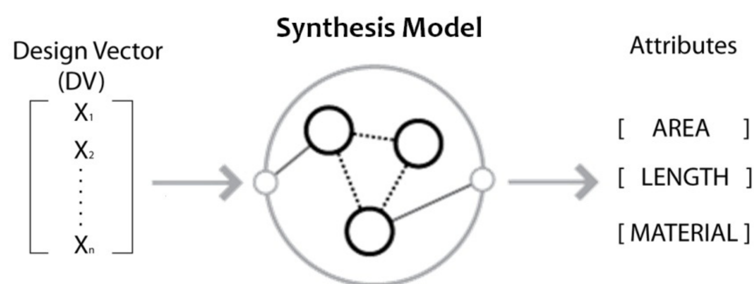
minimizing the search space and leading to feasible solutions.

5.3.2 The Synthesis Model Structure

Synthesis models are constructed via a number of components and modules that represent operations and algorithmic procedures. They also require a type of representation, specifically for the geometric attributes of the artifact. The input to the synthesis model is a design vector. Both the design vector and the structure of the synthesis model affect the nature of the solution space. Within the MDDS framework the synthesis model is expected to output a solution and certain attributes that then become the input to the analysis models (figure 5.2).

figure 5.2 :

Expected input and output of the synthesis model



5.3.2.1 Algorithms in design

An algorithmic approach to design is a systematic encapsulation of design thinking to express the design process. A design algorithm is an articulation of either a strategic plan for solving a tractable problem, or a stochastic search towards possible solutions to an intractable problem (Terzidis, 2006). Describing design processes through procedures is a rationalization process. These descriptions require clearly defined objectives and design languages (Yessios, 1975).

Design can be expressed via different representations, and so are synthesis algorithms. Some algorithms are better expressed as numbers while others are better presented graphically.

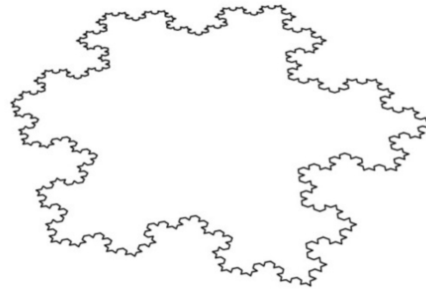
As discussed previously, design is implemented in an iterative process. A specific type of iteration is known as recursion. Recursion is a form of repetition of a local condition where iteration is a more general repetition.

Recursion algorithms are typically known as those that "call" themselves until a stopping condition is found. Repetition should not

be viewed as the reappearance of elements or attributes, but rather the re-implementation of algorithms. A famous example of a recursive algorithm is the Koch curve (Mandelbrot, 1983) (Figure 5.3).

Figure 5.3:

Koch curve is a recursive synthesis algorithm



As stated earlier, designing via algorithms is a rationalization process which forces designers to structure their thinking around causal relationships and the sequence of tasks. This process can be used to devise new designs as well as express existing ones (figure 5.4).

Structures of algorithm vary based on the design problem at hand. The most important fact is that the problem must be describable. Mapping methodologies of problem components, from mere information to algorithmic procedures within the synthesis model, is a design skill.

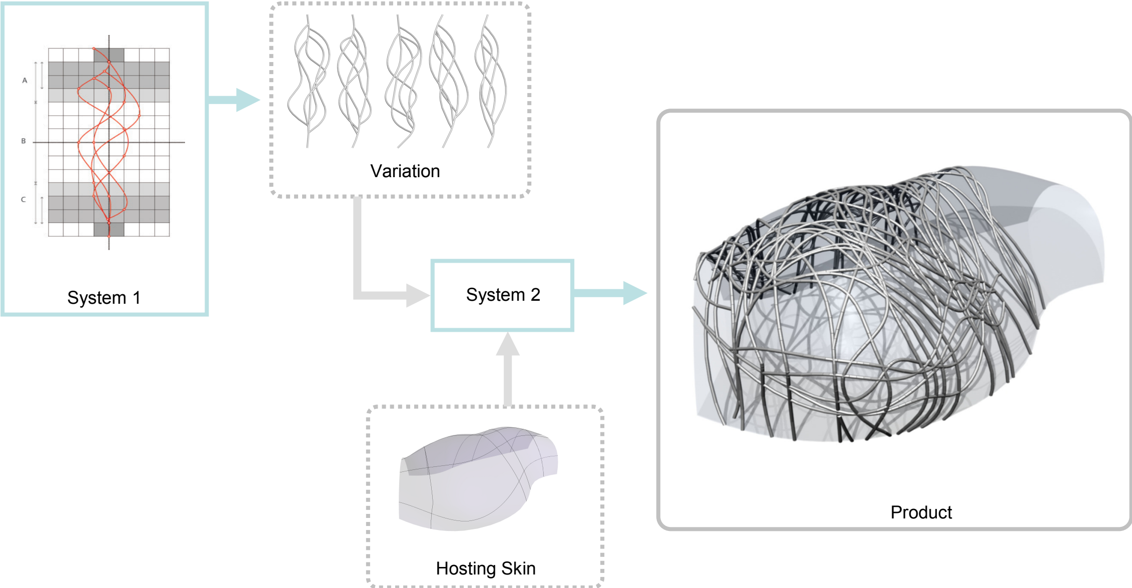
Using Synthesis algorithms, concepts and processes that are seen as inconceivable, unpredictable, or simply impossible by a human designer can be explored algorithmically by computers. However, to achieve their intended goals, synthesis algorithms must be time efficient, deployed within a tractable scope, and implemented to achieve clearly described design intent.

5.3.2.2 Parameters

In mathematics, parameters represent constants in equations that vary in other equations of the same general form. For example, in the equation of a curve or surface, parameters can be varied to represent a family of curves or surfaces. In geometry, parametric equations define shapes (curve, surface, etc.) without assigning direct connections between the coordinates of its points, but rather by expressing these coordinates in relation to one or more independent parameters (figures 5.5).

Figure 5.4:

An example of a synthesis algorithm that generates variations of components that create a structure.



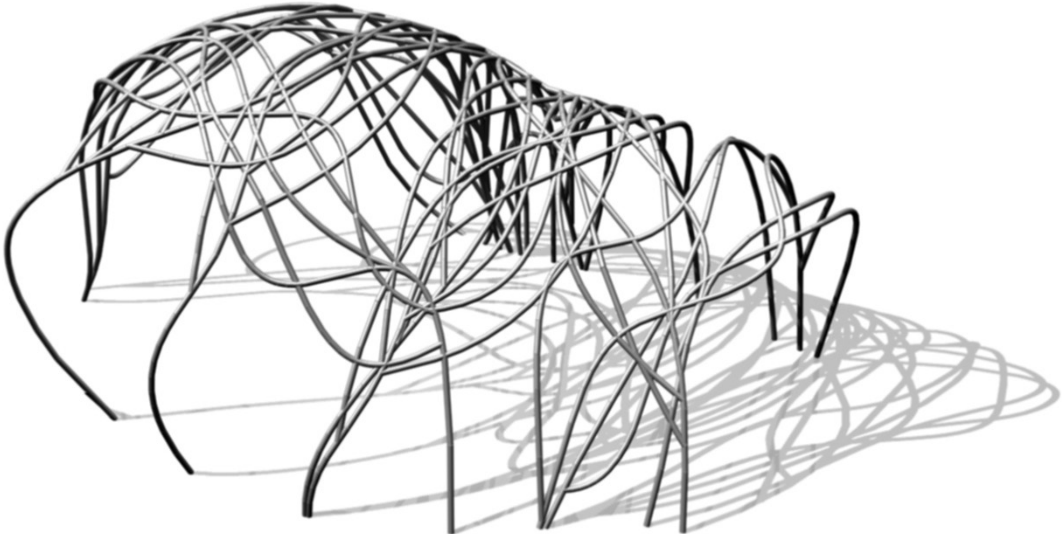
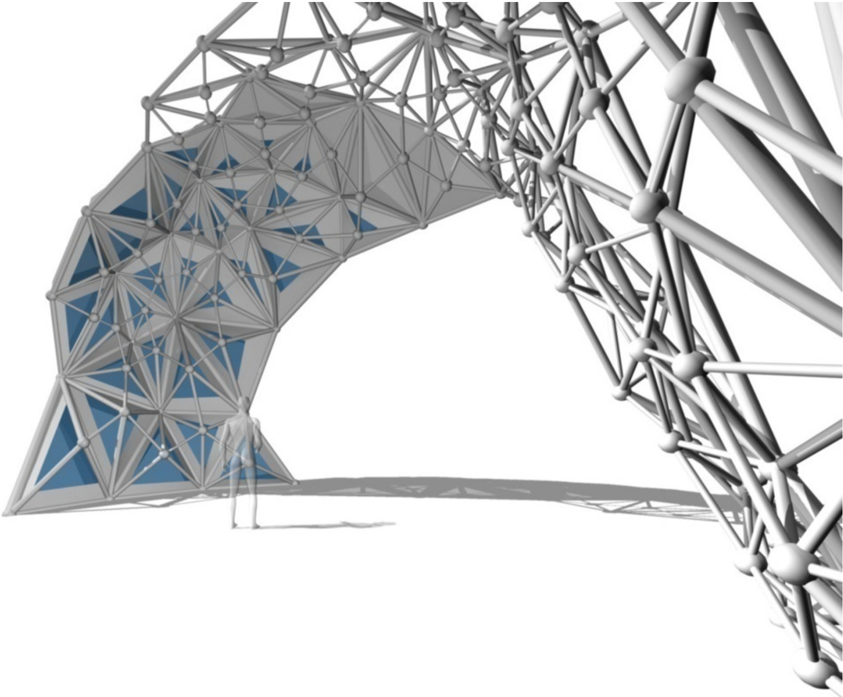
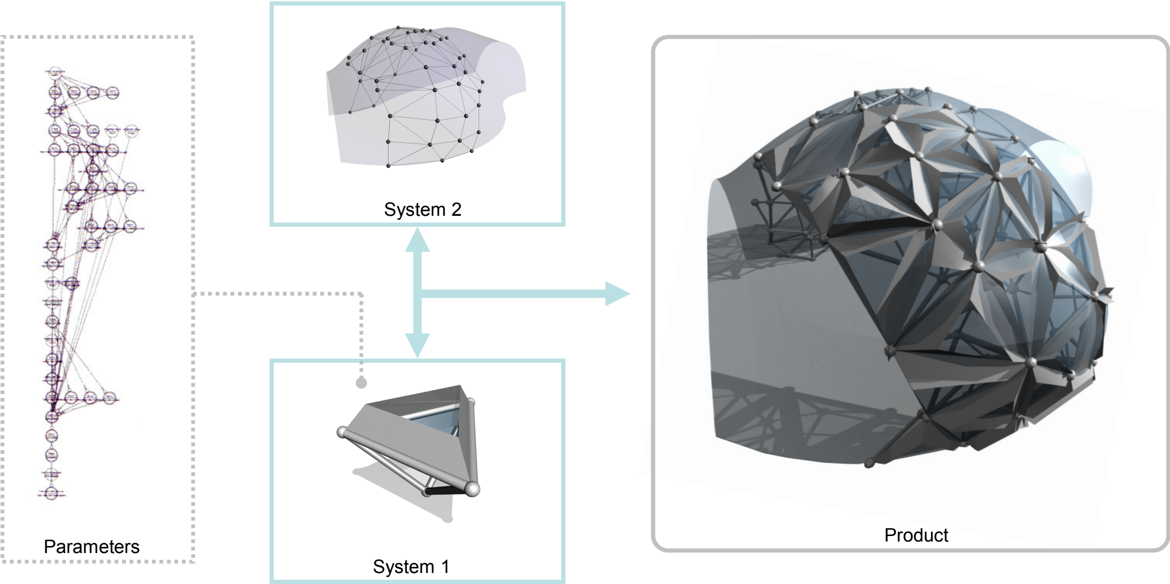


Figure 5.5:

A space truss is generated using a set of parameters





The computer science implementation of parametric equations requires those parameters to hold values before performing a computation. For example, describing a shape with equations is not enough to generate a graphical representation of it.

Describing relationships in terms of parameters is referred to as Parameterization. Since parameters are expressed in numerical values, anything that can be represented by numbers can be parameterized. Parameters can be variable or be fixed. Parameters can be driving numerical values, or a set of algorithms, or geometric elements. They can also be dependent or independent. Dependent parameters are those explicitly defined in terms of other elements. Independent parameters are those that require direct input.

Parameters can act as an interface between input methods and the internal model components. In a synthesis model, parameters may represent value ranges, Boolean conditions, strings, or even algorithms. In this sense, any design component can be parameterized.

Parameters offer control over equations, artifacts properties, or even calculations. This provides the designer of the synthesis model with the ability to manipulate the embedded algorithms and generate a variety of solutions.

5.3.2.3 Design Relationships

We established that synthesis models are structures of operations and algorithms that drive the generation of artifact designs. These designs should facilitate, perform or express a set of clearly defined design intents. These models can be controlled via parameters that expose the model's internal components allowing for control over models behaviors. However, parameters do not provide control constructs within an algorithm. Relationships are ways to control the behavior of the synthesis model. Relationships include: associations, constraints and rules.

Associations

Earlier we introduced the concept of independent and dependent parameters. Designers can set relationships within a model to generate internal feedback loops. Communication among synthesis model components is possible through associations. Associations enforce relationships between components such that one drivers the other. For example, a relationship that expresses the value of $X = Y+1$, is a relationship that associates the value of X to be always

higher than that of Y by one.

Associations can generally be viewed as two types: bi-directional and mono-directional. Bi-directional relationships allow information to flow in both directions. Mono-directional relationships enforce Parent-Children hierarchies that propagate information only in a top down fashion.

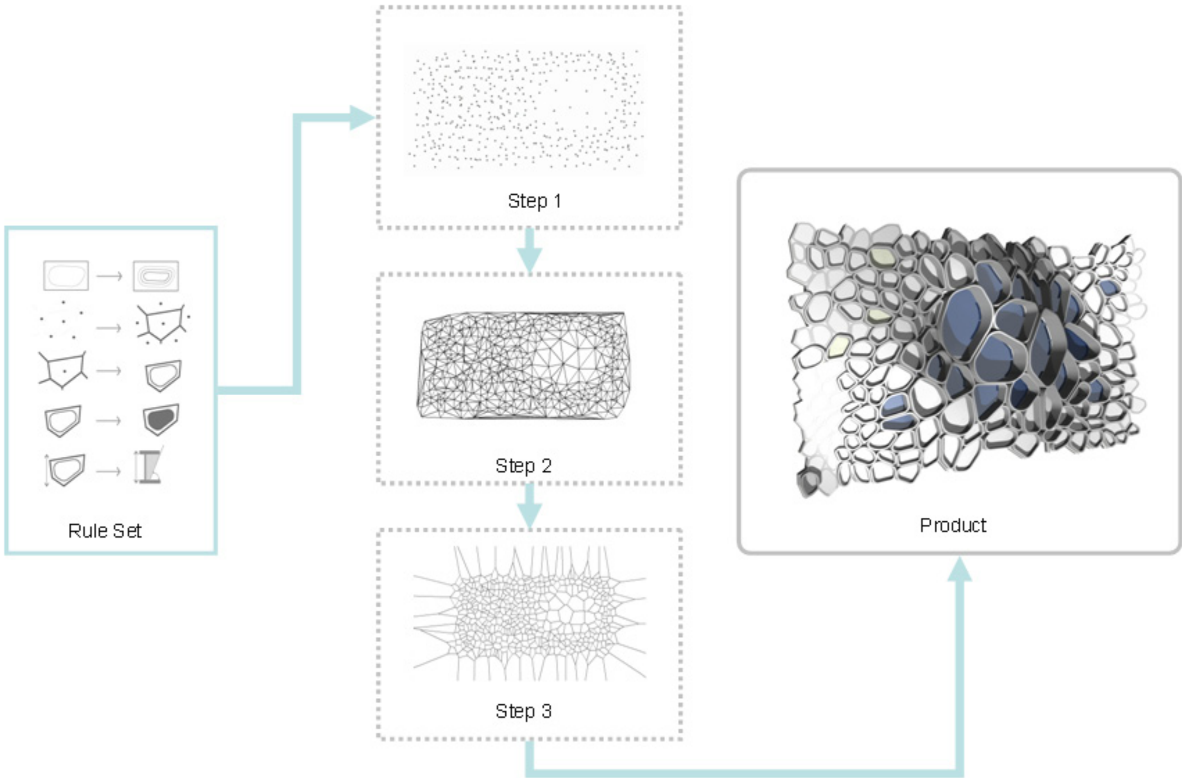
Constraints

Designers and engineers deal with constraints in every design. Constraints are conditions that must remain satisfied for a synthesis model solution to be feasible, thus they can be used to ensure a specific model behavior. Constraints are mainly numerical. For example: the value of X cannot exceed 5, or the angle between line-A and line-B must remain within 45 degrees.

Rules

The concept of using rules within design is not new. Vitruvius's (c. 28 BC) *Ten Books on Architecture* are known to be the first document account of design rules. Vitruvius's work came in the form of recipes to describe roman architectural, engineering and city planning designs. His work had a great influence on Renaissance architects and revived fascination with Roman culture and classicism in subsequent periods. Andrea Palladio's (1570) work also took the form of recipes that described various buildings' parts such as columns, vaults, domes, spatial layout, etc. In his Four Books on Architecture, Palladio provides explicit instructions on how to construct a "proper" Ionic column: "To form the capital, the foot of the column must be divided into eighteen parts, and nineteen of these [same] parts is the height and width of the abaco, half thereof is the height of the capital with the volutae, which is, therefore, nine parts and a half; one part and a half must be given to the abaco with cimacio, the other eight remain for the volutae, which is thus made." (Kalay, 2004).

Nowadays designers from many fields such as architecture, engineering, computer science and artificial intelligence are developing methods and techniques to rationalize synthesis processes in design. But, just as designers previously approached solving synthesis problems, many of these methods rely on heuristics to build the design rules. Building these rules involves knowledge engineering where designers encode a series of facts, preferences, conditions or circumstances within the design rules (Kalay, 2004)



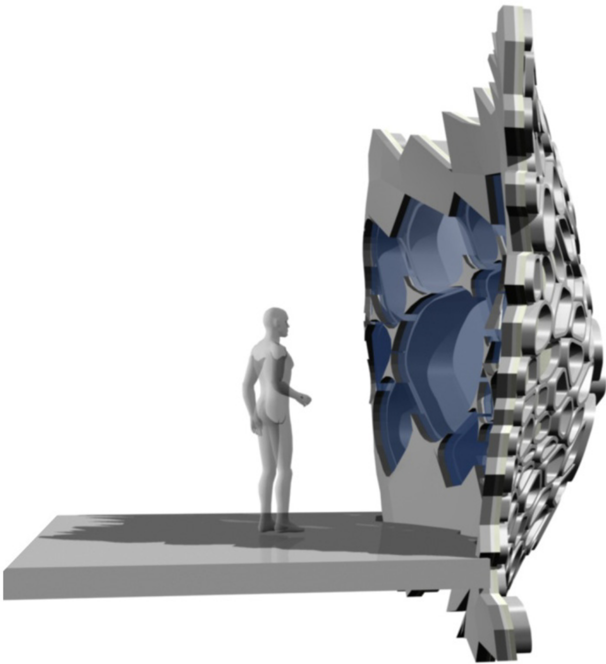


Figure 5.6:

A building skin generated from a set of synthesis rules and an algorithm that generates a Voronoi diagram

Building design rules defines a mode of operation. Designers rationally analyze and critically unpack relationships and dependencies within a design problem to be able to describe it via a set of clearly defined rules. As the same time, rules offer an opportunity to inspect the logic embedded within a synthesis model.

Rules can facilitate some control on the synthesis model and can help regulate the design progress. In addition, while the synthesis model attempts to fulfill the design rules, interesting design solutions may emerge (figure 5.6).

Within a computational synthesis model, the typical form for embedding knowledge is constructing an IF-THEN condition-action model. The IF portion describes the condition in which the THEN part can be triggered. The THEN part includes the action description, which in the case of our model is the design synthesis algorithm. However, conditional statements are not limited to IF-THEN but also include other forms: DO-UNTILL, or WHILE-LOOP, or FOR EACH-NEXT, etc.

Generally, within a computer program, the rule application is performed by a control mechanism known as inference engine. This engine deals with reasoning about current conditions by deductive or abductive methods. In deductive reasoning (also known as "forward" reasoning) the inference engine searches for a rule whose premise (IF part) provides facts. These facts are then added to the system overall repository of facts. In abductive reasoning (also known as "backward" reasoning), the inference engine chooses a specific result and attempts to "prove" it as a conclusion that can be derived from other known facts. In other words, inference engine looks for the THEN-part, and adds that to the overall system facts (Kalay, 2004). An example of a forward chaining reasoning is a production system.

5.3.2.4 Formal Grammars

In the previous section relationships and rules were discussed as part of the synthesis model. This section will introduce well established formalisms that are intended to capture design intent and relations. These are known as formal grammars and include: L-systems, Graph Grammars, Cellular Automata and Shape Grammars among others.

The term formal grammar originated from Chomsky's work on linguistics in 1956 (Chomsky, 2002). A formal grammar is a set of instructions for sequencing a set of symbols to form valid words. The set of all words generated by a grammar formulates a language.

Building vocabulary is similar to mathematical modeling for it entails describing sequences of symbols and operations. The study of formal grammar properties in mathematics is called formal language theory.

Formal grammars, do not only provide instructions to synthesize (generate) strings (concatenations of symbols) in a language, but also determine if a given string belongs to a language through analyzing its internal structure. In computer science, such a process is called parsing.

Grammars demonstrate a robust structure for processing information as they can pack logic of a whole language, and generate its entire set of solutions.

A synthesis grammar language is typically expressed as G = {V, R, S}. The grammar G is a model that includes: a set of vocabulary V, a set of rules R and a set of initial states S.

The set of vocabulary V is expressed via a certain representation. The notion of symbol manipulation in formal grammars indicates that they deal with clearly defined vocabulary which is not limited to alphabets. In general terms, a symbol in a vocabulary is a representation of an element.

The rules set R includes conditional constructs (IF-THEN, DO UNTILL, etc) that fire replacement algorithms. Formal grammars sequences (instructions) manipulate symbols by a process of replacement and therefore can be treated within a computer program as production systems. Replacement rules are typically expressed in the form of X→ Y, which means IF X is found, THEN it should be replaced by Y. Replacement rules can be sequenced in many fashions such as stochastic, procedural approaching a certain state, or recursive where a rule keeps invoking itself until a certain condition is achieved (Mitchell, 1990).

Synthesis grammars also require an input which is a set of initial states S. The initial states set includes the left side of design rules. Initial states define the nuclei that can be used to initiate a solution and resemble the left side of design rules.

Building a synthesis grammar is very similar to devising design processes and rules. Designers tend to formalize and sometimes standardize ways of dealing with design issues. This comes in the form of sequencing a set of actions, and defining characteristics of the final outcome (language). Thus, design grammars enable the designers to critically evaluate design problems, unpack relationships

and consequently build a clearer understanding of methodologies for solving them. Some of the other advantages are externalizing standards, or design criteria to ensure a certain level of quality control over the generated solution; and the ability to generate options that the designers can compare and select from.

Formalisms that will be discussed in this section include: Lindenmayer Systems, which originally was developed to model plants (Prusinkiewicz and Lindenmayer 1991) but has been used in many other fields for design purposes including robotic design (Hornby and Pollack, 2001); Graph grammars, which have been used in many industrial design domains(Alber, 2002); Cellular Automata, which have also been used in a variety of domains including building design and city planning (Batty, 2005); and Shape grammars, which were used in the generation of buildings (Stiny and Mitchell, 1978; Downing and Fleming, 198), and Product Design (Agarwal and Cagan, 1998).

A) L-Systems

Based on Thue's concept of rewrite systems, Aristid Lindenmayer introduced L-systems in 1968 as a method to describe and simulate growth of multi-cellular organisms, typically plants. L-systems perform two main tasks: representing (packaging) information in symbols and interpreting those symbols as growth patterns. The elements (symbols) that exist in an L-system are called axioms. The initial string is a composite of axioms (need not include all axioms). L-systems operate by replacing symbols with one another based on replacement rules. Rules rewrite input strings sequentially. Each step of rules execution represents a generation. Expressing generations (outcome) in an L-system is analogical to providing instructions of how a solution unfolds as opposed to providing blueprints that describe every element in the final solution (Hemberg, 2001).

Strings generated by L-systems can be interpreted as topological maps for they describe connectivity relationships between generations across a production. Mapping the symbols replacement process as connections linking nodes portrays a typical tree structure. If the mapping was interpreted geometrically, one can view a tree like structure. However, L-systems should not only be defined as generators of tree-like geometric objects. Symbols in L-systems are typically maps of instructions where one triggers the other.

The most widely used implementation of L-systems as instructions calculator is that of Turtle Graphics by Prusinkiewicz. In a Turtle

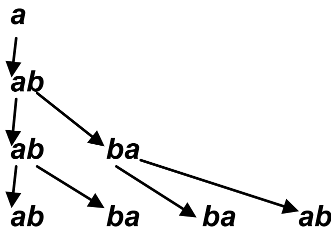
Graphics system, an object (turtle) moves forward, backward, left or right by interpreting generations of strings (symbols) that were created by implementing a number of replacement rules. Furthermore, L-systems are implemented in parallel as opposed to sequential (Hemberg, 2001). The nature of rule implementation in an L-system makes it hard to hand-make the system produce a specific result. A typical rule representation in an L-system is shown below.

The following rules replace "a" with "ab", and "b" with "ba".

a → a b

b → b a

if started with the symbol *a*, produces the following strings,



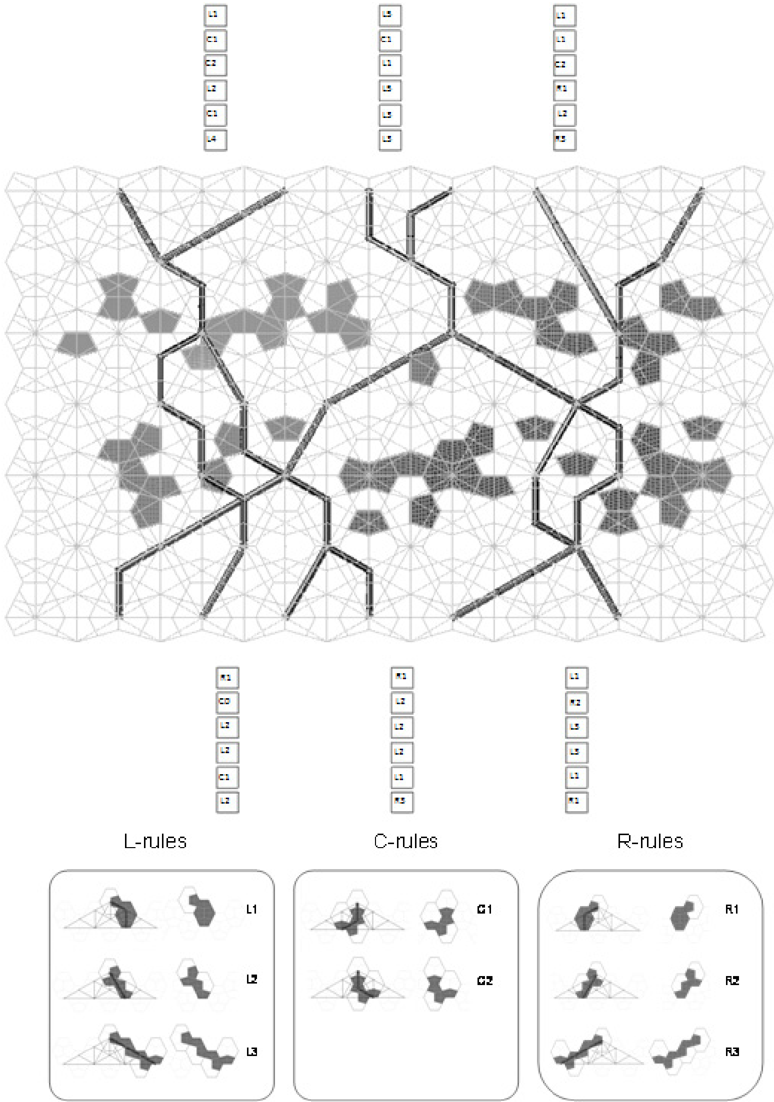
The implementation of the current rules set applies to all symbols across a generation. Thus, the rules are applied in parallel and not sequentially (figure 5.7).

Rules application in an L-system can be guided by parameters that determine which one to execute. This type is known as Parametric L-systems. It differs from basic L-systems in that the production rules have parameters that can hold algebraic expressions. In a Parametric L-system, rules consist of three components: the predecessor, the condition and the successor. For example, a production with predecessor A(n_0, n_1), condition $n1 > 5$

and successor $B(n_1 + 1)cD(n^1 + 0.5, n_0 - 2)$ is written as:

$$A(n_0, n_1): n_1 > 5 \rightarrow B(n_1 + 1)cD(n^1 + 0.5, n_0 - 2)$$

A production matches a module in a parametric word *iff* the letter in the module and the letter in the production predecessor are the same, the number of actual parameters in the module is equal to the number of formal parameters in the production predecessor, and the condition evaluates to true if the actual parameter values are substituted for the formal parameters in the production (Hornby and Pollack, 2001).



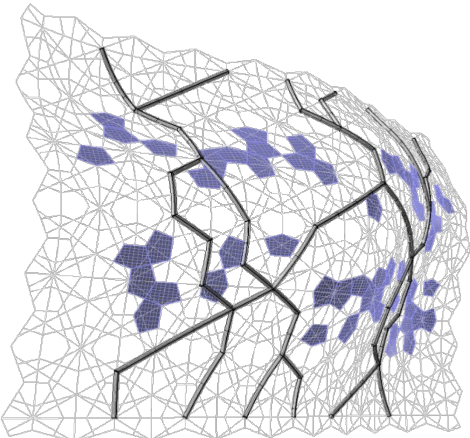


Figure 5.7:

A building skin structure and materiality generated using a set of L-System rules

For example , the PL system,

$$
\begin{aligned}
&a(n) : (n > 1) \rightarrow a(n-1)b(n) \\
&a(n) : (n \leq 1) \rightarrow a(0) \\
&b(n) : (n > 2) \rightarrow b\left(\frac{n}{2}\right) a(n-1) \\
&b(n) : (n \leq 2) \rightarrow b(0)
\end{aligned}
$$

When started with (4) , produces the following sequence of strings,

$a(0)$

$a(3)\ b(4)$

$a(2)b(3)b(2)a(3)$

$a(1)b(2)b(1.5)a(2)b(0)b(1.5)a(2)$

$a(0)b(0)b(0)a(1)b(2)b(0)a(1)b(2)b(1.5)a(2)$

$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(1)\ b(2)$

$a(0)b(0)b(0)a(0)b(0)b(0)a(0)b(0)b(0)a(0)\ b(0)$

B) Graph Grammars

Graphs are well suited for representing technical design objects for they deal with symbols. They are commonly used in modeling knowledge in many engineering applications. They are used to create network graphs, petri nets, part-occurrence trees and class-diagrams in software engineering etc (Alber, 2002).

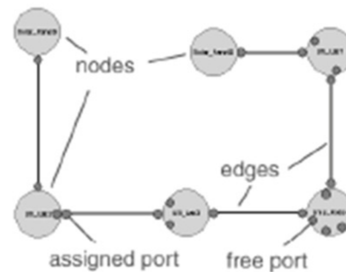
A a graph *G(N,E)* consists of a set of nodes *N* and a set of relations $E \subseteq N \times N$ called edges, whereby the graph nodes as well as the edges have a label assigned to each. Nodes can include information such as attributes or constraints. Constraints define, or rather filter, the set of relationships between nodes by providing information on what can be considered as valid relationships (Alber, 2002).

The notion of nodes attributes and constraints leads to the concept of ports, which are highly used in engineering modeling (Heisserman et al., 2000). A port acts as an interface between nodes providing connectors and rules for valid relationships. Figure 5.8 shows a graph formalism. In this illustration ports are symbolized by the smaller circles which in case a port is unused, reside inside a node or in case it contributes to a relation is aligned to the respective edge.

Graph grammars are similar to other types of synthesis grammars in that they are composed of an initial vocabulary V, and a set of rules R, and initial states S. The vocabulary consists of labeled and attributed nodes. The initial statess_i is any structured combination of the vocabulary elements. A specific axiom or initial state together with an ordered set of rules out of define a production system and corresponds to one specific graph which can be constructed following this program. Table 5.1 gives an overview of the analogies between formal languages and graph languages.

Figure 5.8:

The graph formalism (Alber, 2002)



During the execution of a production system the initial graph s_i is modified by the graph rules thereby evolving in several stages and forming the graph *evolution sequence* $\{G_i^0, G_i^1, G_i^2, \ldots\ldots, G_i^n\}$ with$G_i^0 = s_i$.

Graph grammars deal with attributes and constraints dictating what links may be valid. They also deal with modifying, editing, removing elements from the graph structure when inserting new nodes.

Table 5.1:

Analogies between formal and graph languages (Alber, 2002)

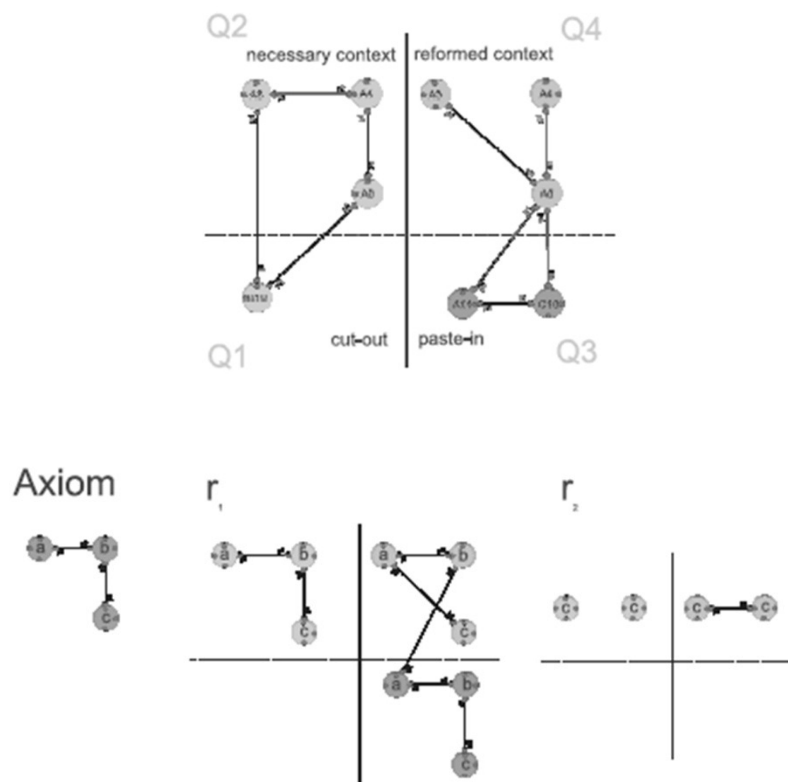
	Formal language	Graph language
vocabulary	Symbols $V = \{a, b, c, \ldots.\}$	Graph nodes $V = N$
rules	Substitution rules $R = \{a \to ab, b \to bcb\ , \ldots..\}$	Graph rule R
Initial states	start symbol sets $S = \{a, aba, ba, \ldots.\}$	Start graphs S
Production system	1 start symbol set + ordered list of substitution rules	1 start graph + ordered list of graph rules
sentence	Structurized symbol set $abbacbabba$	graph
language	Entirety of producible sets	Entirety of producible graphs

T The syntax for a graph modification rule can be represented by an arrangement of graph elements (nodes and edges) in a 2D plane divided into 4 quadrants Q1...Q4 as illustrated in figure 5.9. The two leftmost quadrants Q1 and Q2 contain the conditional part of the

graph rule Graph G_{cond} . At execution a rule a search is performed to check if there exists any subgraph isomorphism between G_{cond} and the actual graph G^k. If an isomorphic subgraph is identified, the generative part of the rule is executed. The modification of the graph is described by the elements contained in Q1, Q3, and Q4, thus, giving the Elements in Q1 a role in the conditional as well as in the generative part of the rule (Alber, 2002).

Figure 5.9:

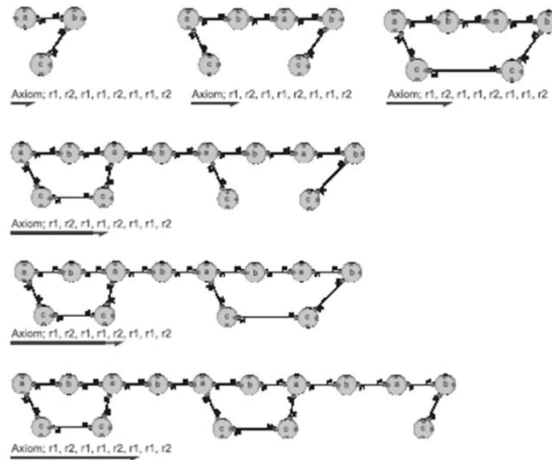
Graph Rules (Alber, 2002)



Nodes in Q1 mark deletions and will be removed from $G_{matc h}$ together with all edges leading to these nodes. The graph nodes contained in Q2 and Q4 play the role of a context in order to specify the embedding of nodes which will be cut out of or pasted into $G_{matc h}$. Hereby the contents of Q2 and Q4 are identical in their nodes but not necessarily in their edges. In this way Q4 can be used to furthermore define a rearrangement in the connection topology of $G_{matc h}$ by removing or adding edges between the identified nodes (figures 5.10 and 5.11) (Alber, 2002).

Figure 5.10:

Graph generation by a production system (Alber, 2002)



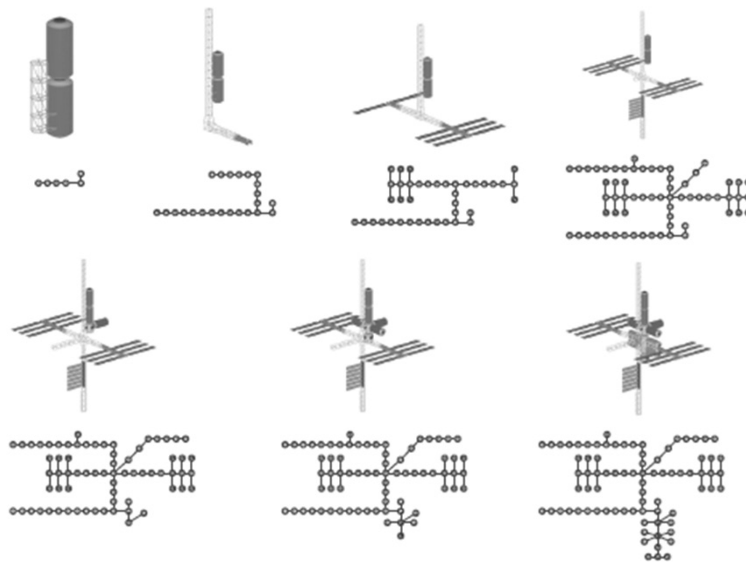
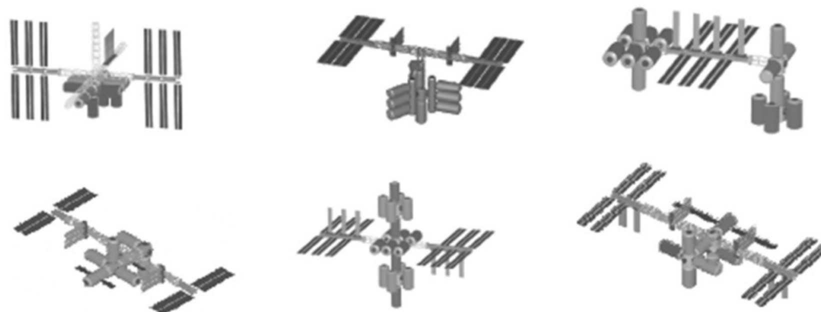


Figure 5.11:

Expansion of a grammatically defined sentence and the corresponding object (Alber, 2002)



C) Cellular Automata

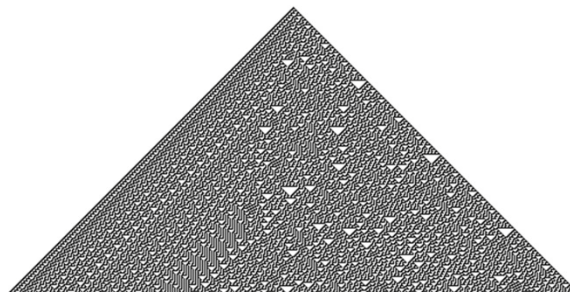
A cellular automaton (plural: cellular automata) is a collection of cells organized in orthogonal grids, each with a finite set of states (usually denoted as colors). This collection of cells evolves over discrete time steps based on the notion of neighborhoods.

A cell changes its state based on its current state and its neighboring cells states following rules. A solution in CA is generated once every cell in the collection runs the embedded rules. CA are sequential, meaning the behavior (state change) in each cell depends on how its neighbors behave. Unlike L-systems were rules are applied in parallel. The first documented account of cellular automata was noted by von Neumann. Neumann was trying to build a model of reproduction where a system rebuilds itself continuously based on embedded rules. Each part in the system has the same set of embedded rules like every other component, thus system administration is local to each. One of the famous examples on Cellular Automata is the game of life, designed by John Conway. Cellular automata proved applicable to many engineering, mathematics and biology domains (Wolfram, 2002).

The simplest type of cellular automata is linear, known as elementary CA. Each cell in elementary cellular automata has two states, black or white. To calculate the number of possible neighborhoods of three cells, we raise the number of states to the number of cells in a neighborhood, so 2 ^ 3 = 8 types of neighborhoods. To calculate the number of possible combinations of 8 neighborhoods with two states, we raise the number of states to the number of neighborhoods, 2 ^8 = 256 possible combinations of neighborhoods of 3 cells each and 2 states per cell. The total number of possible neighborhoods is known as possible CA rules.

Figure 5.12:

Rule 30 cellular automaton



Rules in elementary CA are represented as arrays of back (1) and white (0) unites. They are labeled by calculating the locations of black cells on a binary scale of 128 64 32 16 8 4 2 1. In these location, any black cells. So the representation for a rule 30 is as follows

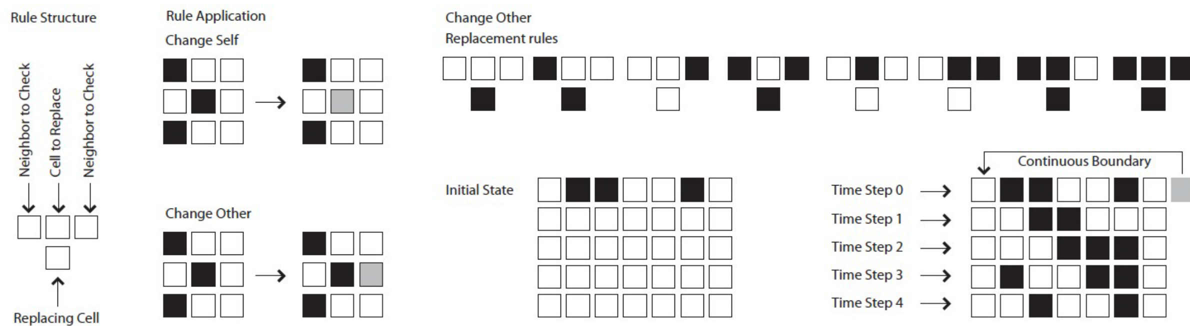
00011110 → (0*128) + (0 * 64) + (0*32) + (16*1) + (8*1) + (4*1) + (2*1) + (0*1) = 0+0+0+16+8+4+2+0 = 30.

Figure 5.12 shows the result of rule 30.

Cellular automata are known to demonstrate four types of behavior: fixed point, periodic, chaotic and random. These types are defined based on the pattern of occurrence of certain behaviors over a defined time period. Most of Cellular automata behavior is known to be periodic, and random. Chaotic behavior, which is viewed as a sign of performing universal calculation, is very limited. This bounds the implementation of CA within a design context to being very limited. Another reason to the limited implementation of CA in design is that it is very hard to predict the outcome of a CA because the system evolves sequentially based on how neighborhoods interact. However, the very same nature of CA behavior makes them reasonable to model and simulate complex systems that result from an aggregate of local simple interactions at the cells level. Below is a diagram showing how CA rules are typically applied.

Figure 5.13:

Rule application of a CA



D) Shape Grammars

Invented by Stiny and Gips in 1972 (Stiny and Gips, 1972), Shape Grammars laid the foundations for major research into algorithmic design approaches in the context of design analysis and design synthesis. The use of Shape grammars for design analysis focused on assessing the amount of embedded knowledge in a given design languages; thus the ability to produce variations that belong to the same family of design languages. Shape Grammars have been used mostly in the design of civil architectural. Some of the most famous grammars used for the analysis of architectural design languages are the Palladian Grammar (Stiny and Mitchell, 1978), Wright's Prairie Houses (Koning and Eizenberg, 1981), Buffalo bungalows (Downing and Flemming, 1981) to name a few (figure 5.14).

Shape grammars are a geometrical construct that express production algorithms and rules through basic geometric elements, points and lines. Shape grammar rules can be interpreted as replacement rules for they consist of a left side (initial shape), an arrow noting an operation, and a right side (the result). Shape grammar operations include can be summarized addition, subtraction, intersection, and transformations. Transformations include: translation, reflection, rotation and scale.

In shape grammars, shapes are more of topological structures than geometric representations (Cagan, 2001). Topological elements do not intersect. They may exist in spaces of similar of higher dimensionality. The following table describes the algebra of shapes as Uij, where i represents the dimension of the topological element, and j represents the space that accommodates it. For example the symbol U12 means it is a segment element that exists in a plane space.

U00	*U01*	*U02*	*U03*
	U11	*U12*	*U13*
		U22	*U23*
			U33

Like L-systems, Shape grammars can be also parameterized. A Parametric shape grammar is composed of the following tuple: (S, L, T, G, I). S is the expression of a Shape Grammar rule in the form (A→ B) which typically means: if shape A is found, it is to be replaced by shape B. L is the set of labels. Labels are notations added to the Shape Grammar rules. They are typically represented as dots (can be also colored). They are part of the rules, not the shapes being calculated. T is the set of geometric transformations that build into the Shape grammar rules. G is a set of functions that assign values to rules parameters (attributes) such as height, width, rotation angles, etc. I is the set of initial shapes which Shape Grammar rules use as to start the calculation. Initial shapes are the left side of a Shape grammar rule (Kalay, 2004).

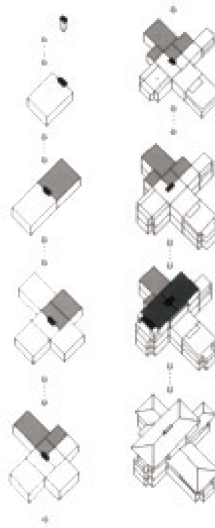
Shape grammar rules that combine various representations are known as parallel grammars such as combining description and shape rules in a grammar. While shape rules are applied to the evolving design geometric shapes, the corresponding description rules are applied to the evolving description. Thus, as the generation of the design evolves, the description of the design is constructed.

Representing design aspects via different combinations of shape

grammars facilitates the manipulation of complex design problems, by breaking them into smaller ones.

Figure 5.14:

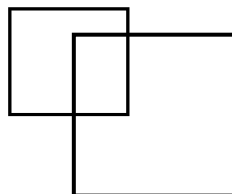
Prairie-style house shape grammar (Koning and Eizenberg, 1981)



A key feature in Shape Grammars is that they are built on the promise of mimicking the process of thinking a designer goes through based on the notion of recognition. For example, a designer may envision, or recognize a certain shape in a complex assembly of lines even though the shape is not explicitly defined (figure 5.15).

Figure 5.15:

In this composition a designer might pick the upper square, the lower one, or the one generated by their intersection

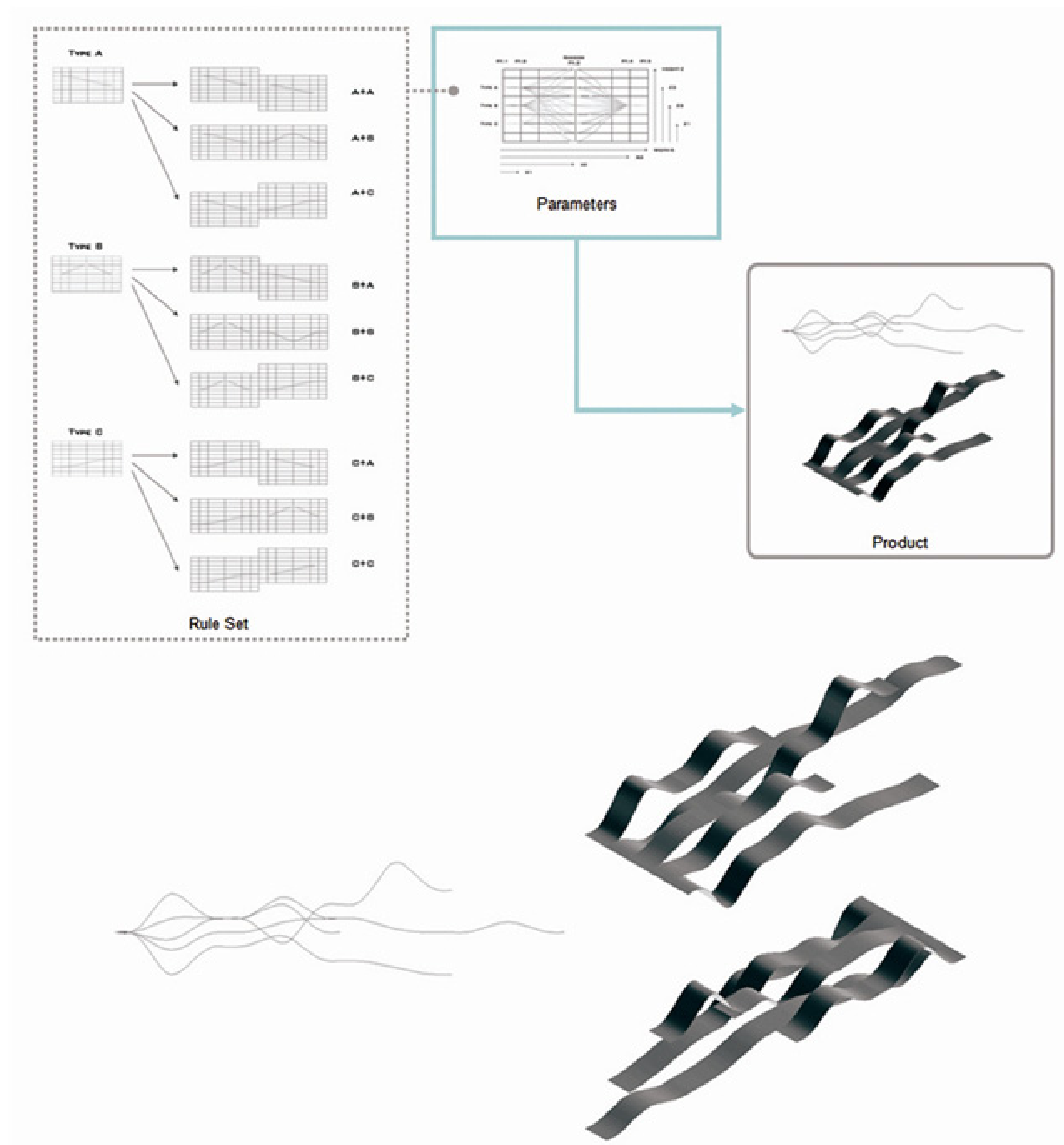


Recognition in Shape Grammars is based on the notion that shapes are non-atomic. They can be decomposed and recomposed freely at the discretion of the designer. Decomposition of elements in Shape Grammars is based on the notions of Embedding and Maximal Elements. Any element is considered a maximal element that includes all elements of similar topology but in smaller size. For example, a line includes all of the embedded smaller lines that can be "seen" inside of it. Thus, if a rule applies to an initial shape, line, it may affect the whole line, or any part of it. This notion of recognition in shape grammars make them virtually unlimited. As long as the system is able to recognize an initial shape that a rule requires, the system will keep running.

Figure 5.16:

A set of Shape Grammar rules can generate many variations of a component

This allows for emergence of new shapes given that designers can pick any shape, edit any rule, and operate in any order. Emergence within this context is the ability to recognize shapes throughout a computation that were not explicitly defined (Duarte, 2001).



However shape grammars do not lend themselves well to computational implementation due to the complications associated with representing shapes numerically as well as the lack of current computational algorithms to recognize emergent shapes.

Nevertheless, Shape Grammars serve as an excellent design development environment as they help formally express design intent through shapes, pushing designer to think in algorithmic, clear terms. This helps casting the design process in a hierarchical fashion where stages and design priorities are expressed and formalized (figure 5.16).

Interpretation

The main difference between the above mentioned formalisms is the need for interpretation. Driven by the design problem, interpretations of manipulated units vary. For example, one might use a CA algorithm to drive the states of units in a neighborhood of cells; then interpret those states as land use (Batty, 2005). In regards to defining the geometry of artifacts, all formalisms mentioned need a mechanism of interpretation between the actual formalism and its interpreted geometry, except for shape grammars. The nature of the shape grammar formalism lends itself well to geometry as it describes design elements as shapes. However Shape grammars can use interpreters for aspects of design synthesis that are not captured by geometry.

5.3.3 Computational Representation of Synthesis Models

Synthesis design models capture the artifacts form attributes. These could include material properties or shape characteristics. The latter will be the focus of this section, and thus, the representation of interest will be mainly geometric.

Representation is a structure of symbols that expresses the environment or design intent through a set of mapping rules (Kalay, 1989). Representations serve as interfaces that define how we interact with and study artifacts properties.

The representation of artifacts in synthesis models requires setting up clear definitions of their composing elements, and the operations that can help implement them. Modeling artifacts is typically partial and so are the representations expressing them.

Geometric representation of artifacts in a computer environment can be constructed in the form of wireframes, surface or solid

representations. Wireframe representations can be thought of as a set of curves that describe space discontinuities. Surface representations are built off of wireframes. They describe two dimensional spaces. Wireframe and surface representations are ambiguous. They do not provide information on what is inside or outside, what is filled or empty. Solid models offer a better depiction of physical artifacts for they are un-ambiguous. They provide information on closure (or water-tightness), boundaries, inside and outside, and well-formedness allowing for automated manipulation and testing (Kalay, 1989). In the rest of this section, geometric representation will be explored through solids.

In the 1950s, numerically controlled machines were introduced at MIT. Because those machines were mainly used in domains like aerospace engineering and automotive design, interest in sculpting and smooth modeling arose. In this period, came the works of Bezier, Coons, Gordon and others in the early 1960s which provided methods to describe surfaces mathematically. In parallel, research into parametric wireframe drafting was also being developed at MIT and the first drafting software that emerged was Sutherland's sketchpad in 1963.

Later in the 1970s, there was a desire to represent artifacts as solids due to the limitations inherent in wireframe and surface representations. In this period, two camps were formulated: the first under Ian Braid in the University of Cambridge who worked on representing solids with bounding surfaces, and the second under Requicha and Voelcher at the University of Rochester who worked on representing solids as Boolean combinations of primitives. Their method was later known as Constructive Solid Geometry (CSG). Later, in 1975, came the work of Baumgart on winged-edge structure as a method to build boundary representations for solids (Shah and Mäntylä, 1995). Following Baumgart's structure was the work of Eastman on split-edge structure at Carnegie Mellon in 1977.

A solid representation is an abstract notion of a symbol that is 1) rigid: has invariant configuration or shape, which is independent of location and orientation; 2) homogeneous in three dimensions: has an interior, connected boundaries with no isolated or "dangling" portions; 3) finite: occupies a finite portion of space; 4) closed: remains a closed water-tight artifact under transformations (translations and/or rotations) or operations that add or remove material (welding, machining); 5) describe-able: has a finite set of faces and edges; 6) boundary-determinable: has an inside and outside (Requicha, 1980).

Solids are typically built by representation schemes. "a representation scheme is defined formally as a relation s: m→ r. We denote the domain of s by d, and the range or image of d under s by v. Any representation in the range v is said to be valid since it is both syntactically and semantically correct (i.e., it belongs to r and has corresponding elements in the domain d)" (Requicha, 1980).

Schemes for representing solids rely on two components: data abstraction and hierarchy of elements (Kalay, 1989). The first component, data abstraction, describes geometrical and topological notions, such as space, surface, line, point, and shell, face, edge, and vertex respectively. Data should be mathematically modeled for three reasons: (1) mathematical models can be studied independently of computational considerations; (2) such important concepts such as representational validity and ambiguity can be defined mathematically; and (3) a rich body of mathematical knowledge can be applied to the study of geometric modeling (Requicha, 1980). The second component, hierarchy, describes techniques for structuring data. Hierarchies define the relationships between the composing elements of solids (abstract data), leading to defining topologies of solids. Hierarchies describe these relationships in a bottom-up fashion. A number of schemes for building solid representations follow:

• Spatial Occupancy Enumeration (SOE) of voxels: space is subdivided into regular cells, and the target artifact is specified by the set of cells it occupies. Models described this way lend themselves to finite difference analysis. This is usually done after a model is made, as part of automated pre-processing for analysis software.

• Cellular Decomposition (CD): similar to "spatial occupancy", but the cells are neither regular, nor "prefabricated". Models described this way lend themselves to finite element analyses (FEA). This is usually done after a model is made, as part of automated pre-processing for analysis software.

• Sweeping: an area feature is "swept out" by moving a primitive along a path to form a solid feature. These volumes either add to the artifact "extrusion" or remove material "cutter path". Also known as 'sketch based modeling'. Sweeping is analogous to various manufacturing techniques such as extrusion, milling, lathe and others.

• Boundary Representation (B-rep): a solid artifact is represented by boundary surfaces. Surfaces definition is based on edges, also

knowing as 'surfacing'. Thus, B-reps are verbose. Each object comes with it explicitly defining surfaces.

• Constructive Solid Geometry (CSG): simple artifacts (primitives) are combined using Boolean operations (union, difference, intersection) and linear transformations (Kalay, 1989).

Schemes for building solid representation may be also combined into Hybrid representations. Examples on Hybrid representations include: 1) CSG/B-rep hybrid, which is used as the basis for the input language of some geometric molding software; and 2) CSG/Sweep hybrid, which useful for the verification of programs for numerically controlled machine tools (Requicha, 1980).

Operations used in manipulating solid representations typically with construction and transformation of shapes. Construction methods describe shapes using Boolean operations (Union, subtraction and intersection); axial sweeps (extrusions) and rotational (revolves); Lofts; among other operations (Kalay, 1989).

Transformations are operations that change one instance into another, while preserving its properties. Geometric properties are those that remain invariant under *isometric transformations.* Transformations are of two types: proper and improper. Proper transformations are translation, scale and rotation. They do not change the artifacts' properties or relationships among the artifact's topological elements (vertexes, edges and faces). Proper transformations preserve angles and scales relationships, but not necessarily distances. For example, scaling an object isometrically changes its size, but not the angles among its faces or the length ratios among its edges. Improper transformations are those that change relationships among artifacts topological levels such as reflection (Mitchell, 1986).
The accuracy of solid representation facilitates automated generation of documentations, and detection of interferences, embedding of artifact information, among other things.

Although solid modeling was a pivotal development in the manufacturing field due to its accurate representation, it fell short in providing ways to regenerate artifacts' representations based on parametric variation. Later, an enhanced type of modeling systems, where designers can access geometry parameters and reorder the stack of operations, was introduced.

Within a Parametric CAD modeling system, parameters act as place holders for numeric values that drive the geometric and topological

structures of an artifact. They offer the ability to regenerate (vs. redraw) new representations of artifacts. Regeneration (or updating parameters) can be viewed at three levels. The first deals with parametric entry. At this level, designers create artifacts by entering a number of parameters through a user interface. The second level deals with parametric editing where users can edit any created artifact at anytime and regenerate new representations by either changing the stack of operations or editing parameters values. The third level offers parametric updating. At this level, users selectively update, edit and link parameters to one another allowing for robust control of parts and assemblies (Sacks et al., 2004). Controlling artifacts attributes via parameters triggers chains of update cycles that reevaluate dependencies while satisfying topological and geometric relationships.

Parametric CAD systems typically offer two types of parametric objects: typed and type-less. Typed objects are usually solids. They resort in libraries of ready-made parameterized geometries that require specific contexts to trigger them. Typless parametric objects typically include: primitive solids such as cubes, spheres, cylinders, cones and torai; and surfaces and wiresframes such as NURBS (Non-Uniform Rational Bezier Splines) and NURBS surfaces. Unlike primitive solids, NURBS describe topology with various geometric representations via degrees of curvatures and orders of polynomials describing them. What makes NURBS curves and surfaces interesting is the ability to easily control their shape by interactively manipulating the control points, weights and knots, and hence the ability to reshape objects as if they were made of an elastic material (figure 5.17).

Parametric modelers evolved to include relationships such as: constraints (of various types); associations (defining a parameter in terms of other parameters or measures), and rules that allowed for additional control over the created parts and assemblies.

Associations express relationships between different geometric entities. As discussed previously, associations can be Bi-directional relationships allow information to flow in both directions. Mono-directional relationships enforce Parent-Children hierarchies. Associations can also be expressed as equations defining a parameter in terms of other parameters or measures. For example, one may define the height of a wall-A as *h* = area of slab *0.5, or *h*= the height of wall-B *1.2.

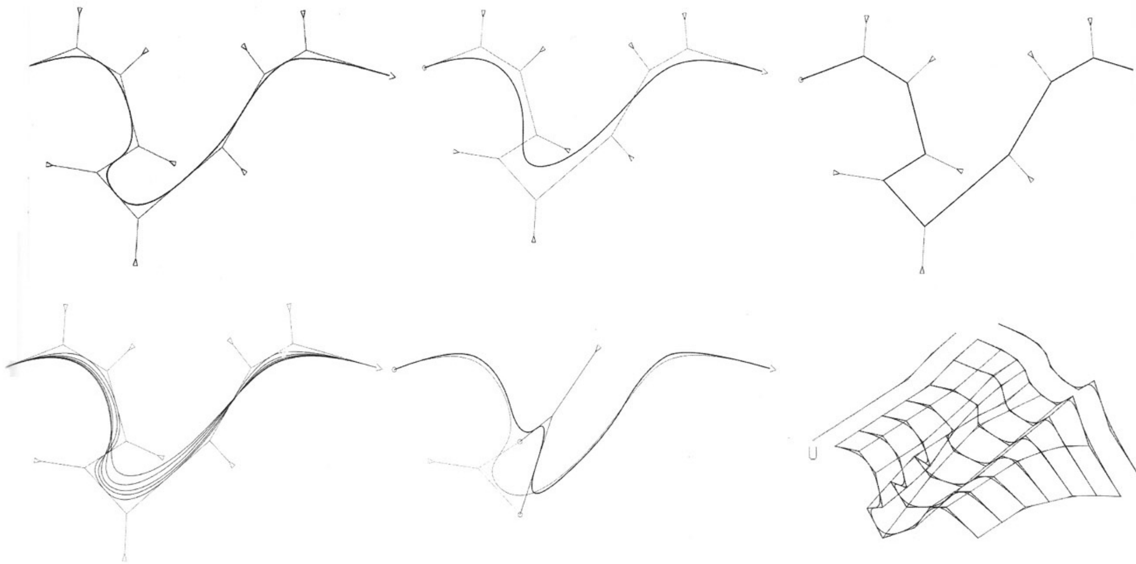


Figure 5.17:

various NURBS can be constructed by the same number of control points with various degrees

Constraints are expressions that define how artifacts' behaves within a defined context (Sacks et al., 2004). Current parametric systems offer three main types of constraints: geometric, engineering, and time. Geometric constraints dictate how two entities relate to one another in space. These include relationships pertaining to location and orientation such as parallelism, perpendicularity, coincidence, offset, rotations, etc. They also include measurements such as distances, lengths, angles, etc. Engineering constraints deal with materials properties, and machining processes. Time constraints are typically applied at assembly levels to control how different parts interact with one another through time (Anderl and Mendgen, 1996). Time constraints aid in: the development of procurement and construction schedules; and the analysis of buildings' behaviors over time. Artifacts can be over constrained, constrained, under-constrained, and un-constrained. Constrained artifacts are those which a designer have full control over their behavior. Parametric modeling systems rely on constraint solvers to offer valid solutions. The number of possible behaviors (solutions) an artifact exhibits is a reflection of the set of imposed constraints and the performance of constraints solvers.

Some parametric CAD modeling systems (such as CATIA) offer additional form of control via rules. Integration of rules offers contextual control leading to embedding knowledge in parametric models. Rules are usually expressed in the form of "if… then".

Embedding knowledgebase modules in parametric models can regulate the development of design solutions where each group of domain experts contributes to the design process. Embedding constraints and rules allows for building robust assemblies leading robust design automation processes.

5.3.4 Modeling Variation

In the previous sections the internal parts of the synthesis model were discussed. These models include various types of representations, rules and algorithms, and parameters which are fixed, associated or varied. In this section we will discuss the input to the synthesis model, namely the design vector. A design vector is the set of model variables that provide means to control the generation of solutions. The number and type of variables included in the design vector affects the generated solution space. The solution space is also affected by the knowledge embedded within the synthesis model it's self.

5.3.4.1 Synthesis Design Vector

Modeling system requires expertise in defining the scope and boundaries of the design intent. Designers never operate on a closed set of requirements, thus design intent may very well vary. In this context, model developers face many challenges such as: Which parts of the system should vary and which ones should be fixed? What are the valid ranges of the different variables? And so on.

The set of variables that are allowed to vary at every design iteration is known as the design vector (DV). Variables in the design vector can represent any component or property in the model. They may include a range of numerical values, a design rules, or even an initial state. The design vector can include a set of elements from the synthesis model that are allowed to vary, and not necessarily all the elements.

The values of the variables in a design vector at a point in time define a configuration. The design solution generated by a certain design vector configuration is an instant and represents a point in the solution space.

Extreme care should be taken while building design vectors to account for problems of generating unfeasible solutions that can occur due to several reasons such as defects in the structure of the synthesis model or the manner in which the design vector maps to the synthesis model.

There can be more than one way to represent the relationship between the design vector and the synthesis models. One way is to consider a one to one mapping between a variable in the design vector and a property in the synthesis model. For example the variable *x* in the design vector can map to a parameter in the synthesis model that represents a component's length. If *x* is varied the length will also vary. This one to one representation is intuitive and easy to implement. However there are several issues with this implementation including scalability and knowledge of the components relationships (Bentley and Kumar 1999). With small numbers of variables there are no major limitations but when scaling up and increasing the number of variables the implementation becomes more tedious and might produce undesirable combinations of the variables that generate unfeasible solutions. This is an aspect that cannot be controlled with a one to one mapping.

Another implementation of design vectors considers a one-to-many mapping. This type of implementation advocates defining a limited number of design variables that can control a larger set of properties within the synthesis model. This mode of implementation relies on wiring using rules a group of internal component properties within the synthesis model that can be controlled by single design variable in the design vector. An example of such an implementation is by imbedding an initial state in the design vector and using a formal grammar in the synthesis model. With every iteration when the initial state is changed, the grammar triggers a chain of reactions internally within the model producing a different solution.

Furthermore, with such an implementation more control over the design solution can be achieved due to packaging of several preferred property combinations and relationships in the synthesis model and hence producing more feasible solutions.

The relationship between design vectors and synthesis models demonstrates a rich area for investigation in almost every design problem. The question of which implementation to follow is highly dependent on the domain of application, level of control desired, design intent, and the designer skill.

5.3.4. 2 Solution Space

A solution space may vary in size or nature depending on the synthesis model that defines it and the amount of variations allowed by the design vector that controls it. Due to their vast size, solution spaces are best explored through automated execution of

computational synthesis models.

Human immediate interpretation or understanding may not comprehend the size of possible solution spaces, or envision the types of solutions possible to be found.

To better explain the notions of solution spaces, consider the following example. An LCD screen offers a fixed number of pixels. Each pixel can carry a fixed number of colors. In today's available technology, LCD monitors can hold colors of 16 bits, meaning that each pixel may have 65,536 possible values (2^{16} = 65,536 colors). In a monitor 1280 x 800 resolution, we can get the following number of display solutions $65536^{1024000.}$. Such a number of possible solutions seem hard to comprehend. Let's consider colors of 1 bit, black or white, and a smaller screen resolution of 640 X 480. The number of possible color combinations is 2^{307200}. This is still a huge number.

The generation of the complete solution space by enumeration is possible only if the design vector is small. For its term, enumeration entails highly intensive number crunching. This is achieved through systematic combinatorial solving of the synthesis model (Kalay, 2004). A graphic representation of all solutions is a tree where branches demonstrate variations within a certain configuration (figure 5.18)

Depending on the design vector, the number of enumerated solutions may possibly be larger than what any computer can calculate. In such situations, designer should consider investing in searching and developing specific areas within a solution space.

Feasible solution space

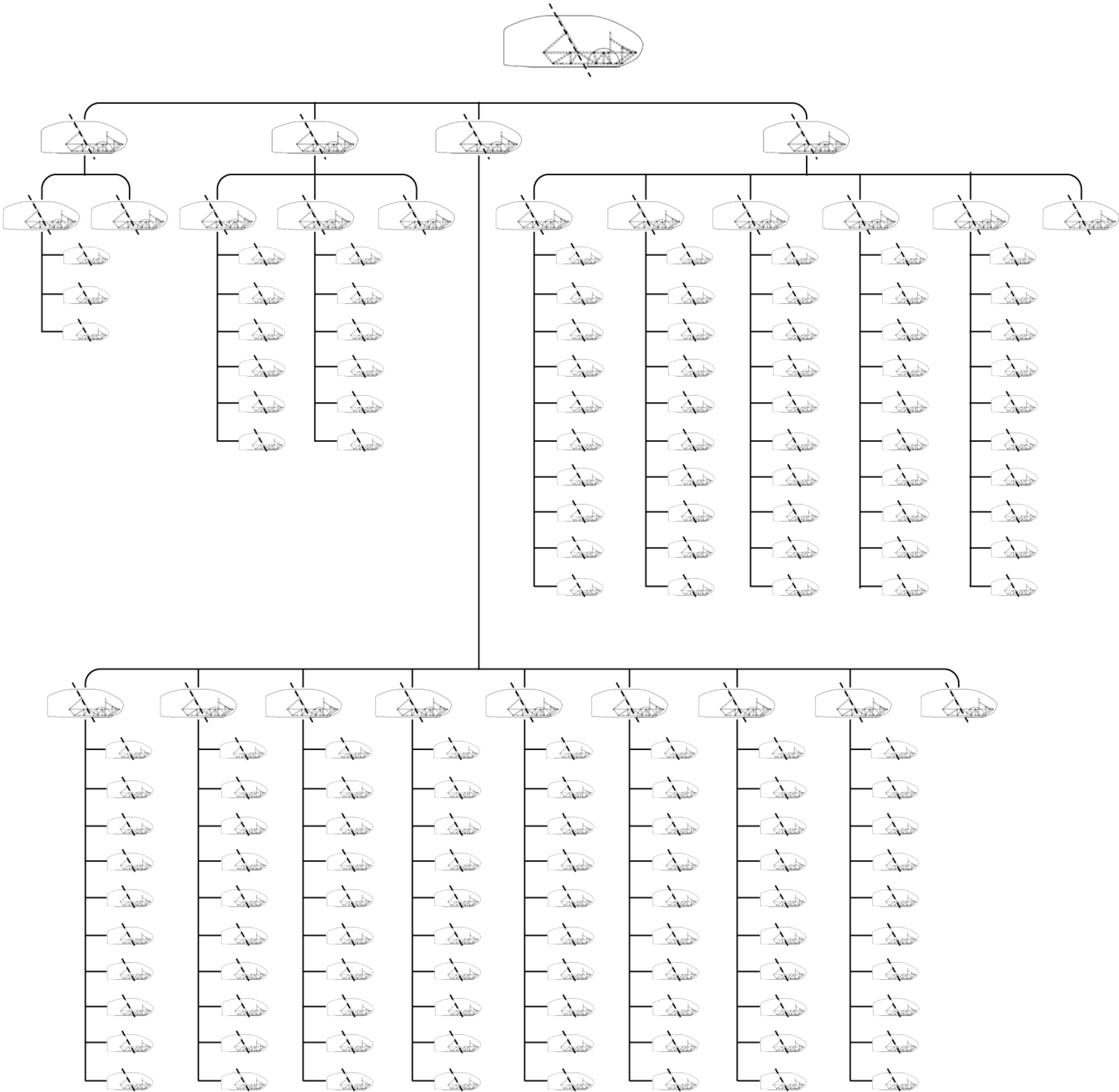
While solution spaces are defined by synthesis models and design vectors, they may offer solutions that do not satisfy certain expectations. This leads to the notion of feasible spaces. The feasibility of solution spaces can be controlled via rules and constraints. Constraints provide means to filter or rather disqualify unsatisfactory behaviors or solutions generated (discovered) by the system. A feasible solution space is one that not only satisfies the synthesis design rules, but also abides by the imposed constraints. An accepted solution is known as a feasible design.

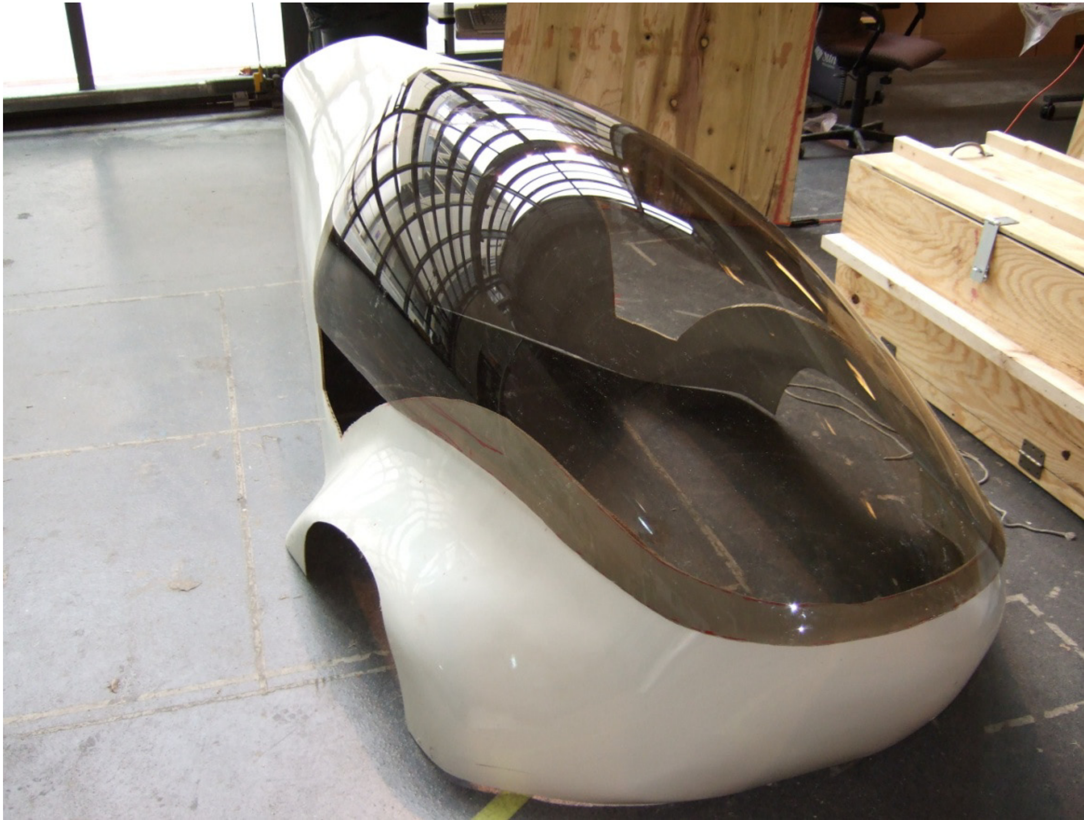
Typically, the larger the solution space is, the higher the possibility of finding valid and possibly novel, designs. However, a smaller solution spaces provides faster navigation and search for qualified solution candidates.

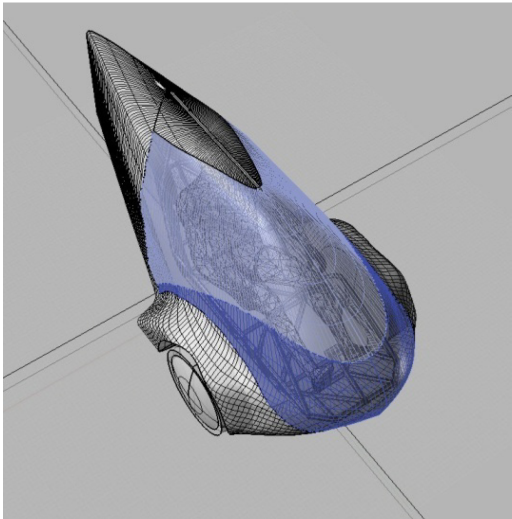
Figure 5.18:

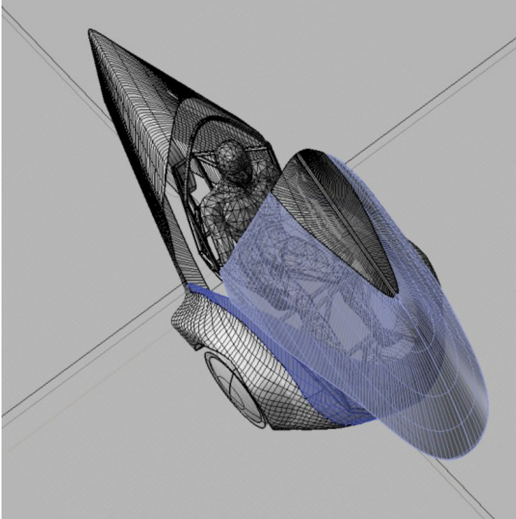
An enumeration tree for an ingress-egress system

Project credits: Anas Alfaris, Nii Armar and Martin McBrien









The problem with a solution space is that it can include a large number of unwanted design solutions. This can be controlled by embedding knowledge and performance evaluation as will be discussed next.

5.3.4.3 Knowledge and Performance Encoding

In the previous sections, rules were mentioned as a construct that allows for embedding functional knowledge in synthesis models. This knowledge is typically used for the assessment of an evolving design solution. This built in assessment of a synthesized design solution serves as a guide for the generation of solutions.

It is important to note that the analysis of a candidate solution could be a time consuming processes (as will be discussed later in analysis models). This is not only a problem of analysis models, but also of synthesis models, which are responsible for generating the low performing solutions in the first place. Avoiding such problems helps built more robust and resource efficient systems. This depends on the designer's experience in building synthesis models.

Knowledge can be embedded in the form of information about the required context within which an artifact will function, or about its internal structure, or its limitations, etc. Such information can be encoded within synthesis models using rules and algorithms to help avoid generating unsatisfactory results.

Mitchell introduced the idea of function modeling in 1991 in a function grammar (Mitchell, 1991). These serve as definitions of possible combinations of functions that a grammar (model) can generate or handle. Fenves and Baker (1987) presented a function grammar for the conceptual design of structures, using architectural and structural critics to guide the design configuration. Similarly, Rinderle (1991) presented an attribute grammar. It included strings describing parametric and behavioral properties of the used symbols. Finger and Rinderle (1989) introduced a grammar for the form and function configurations of mechanical systems, they called bone graph grammar. All of these grammars form an expression, of a function (Cagan, 2001).

Embedding basic understanding of functions in the synthesis model can help minimize the size of the solution space. In the first case, where designs are still at an intermediate stage, integration of functions requires more sophisticated mappings and designing of rules.

However not all functions can be embedded due to their nature such as functions that can cannot be assessed incrementally but can only be assessed after the full solution is generated. In such cases the assessment will be left to the analysis model.

Defining the scope of synthesis models depends on the experience and skills of the model designer. Defining the scope of the model is typically driven by design intent. Design intent can be expressed through functional requirements, which intern can be encoded in the model. These will help guide the generation process of feasible solutions.

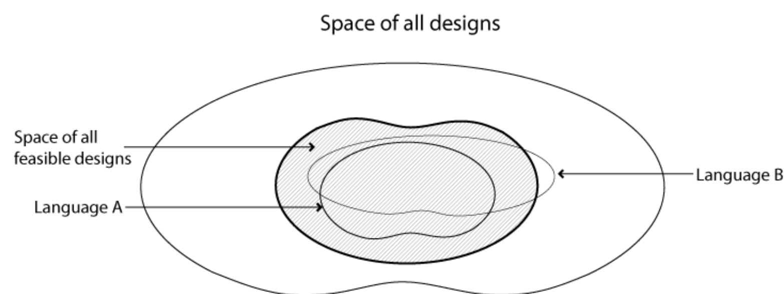
Restrictive vs. unrestrictive systems

Synthesis models can be restrictive (knowledge-intensive) or unrestrictive. While embedding knowledge can guide the synthesis of feasible spaces, they may lead to omitting other unexpected, or rather novel solutions (Cagan, 2001).

Consider figure 5.19 in which the big ellipse shows a design space, and the smaller ellipse-like shapes show solution spaces defined by two synthesis models A and B. Synthesis model A is restricted to produce a solution space that fully falls within the space of feasible solutions. While synthesis model B produces a larger solution space that can have many more interesting solutions, but the space is not fully confined to the feasible design space and therefore might produce unfeasible solutions.

Figure 5.19:

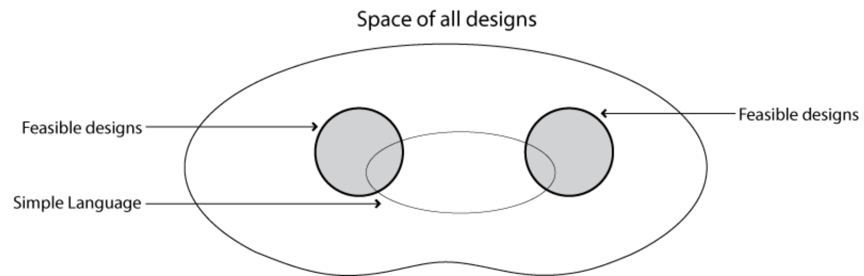
Various synthesis models can generated different solution spaces



The ability of an unrestrictive synthesis model to span large design spaces can help locate interesting and novel solutions that are also feasible but are not part of the original feasible space, although this might require extensive search (figure 5.20).

Figure 5.20:

An unrestrictive model may be able to span several feasible solution spaces



Deciding between a restricting synthesis model and an unrestrictive one is not an easy task. On one hand, restricting synthesis models ensures generating a valid set of feasible solutions that can be better searched and therefore cuts time of generating unsatisfactory ones. On the other hand, building unrestrictive systems may lead to generating unexpected novel solutions that still satisfy the requirements. However because they are less constrained, unrestrictive systems can generate many unsatisfactory solutions.

5.4 Analysis Models

5.4.1. What is an Analysis Model?

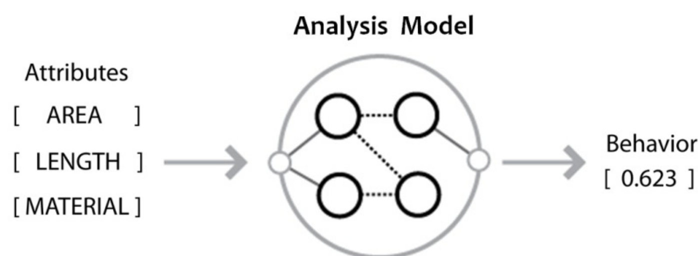
Analysis is defined by Alexander (1964) as the measure of how well a proposed or given design solution fits the goals it is intended to meet. An analysis model in this sense infers certain behaviors from a design solution that are relevant to that specific model or discipline. The model operates on design solution data through laws of physics and geometry to produce the desired rating. It also depends on specialized disciplinary knowledge such as heuristics, formulae, or simulations to determine how this data is transformed into behavior and performance characteristics.

The key issue in the predictive nature of these models is that their behavior is assumed to be analogous to that of the artifact, and thus they are used to study that behavior. This assumed similarity should be based on sufficient understanding of both the artifact and the model that represents it.

In an analysis model the inputs denote the specific attributes under which the behavior of that artifact is examined, while the output defines that behavior (figure 5.21). Those attributes are represented in terms of geometry, parameter values, boundaries, and initial conditions. This thesis will apply mathematical models to predict the behavior of a synthesized system or artifact. These will comprise many types of models, including analytical, numerical, surrogate models and others.

Figure 5.21:

Expected input and output of the analysis model.



Variables, parameters, equations, inequalities, and algorithms in abstract mathematical form are used to represent these models (Jacoby and Kowalik, 1980). The interior of the model is mostly structured and comprises a group of interconnected components that represent an aspect of the artifact or system. The complexity of these interconnections always leads to enormous amounts of computation and information manipulation, requiring the use of computers to handle the models. Selecting the appropriate solution

scheme strongly affects the modeling purpose. There are many challenges and difficulties associated with using computers in analysis models.

Sometimes the lack of sufficient theoretical understanding of the artifact makes it hard to define the mathematical relationships representing that artifact. Assuming that these relationships are even defined, assembling them together may not necessarily build up a problem that is solvable. At the same time, enough data may not always be available for a solution if the problem is presumed to be solvable. There may still be errors that result from the intertwined approximations and computations involved in the modeling process. Other challenges include computational cost and difficulty of validation.

Models can even have overlaps with each model containing a variety of abstract structures. An analysis model will usually be coupled to only some aspects of the phenomenon that is in question. Two models related to the same phenomenon can differ to a great extent. This can result from differences in initial model requirements, conceptual differences or ongoing decisions made along the modeling process.

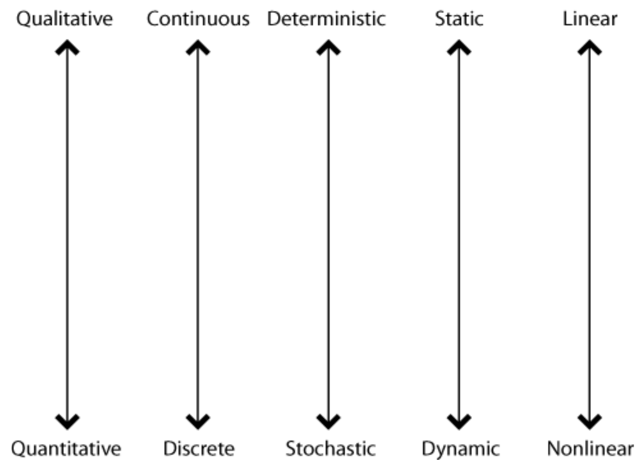
When it comes to selecting a modeling method or deciding on which type of model to construct, there is a wide range of methods that can be employed to model different aspects of the artifact or system. As mentioned earlier, these methods rely primarily on the purpose of the model. This entails determining the required type, level and fidelity of information, in addition to the amount of detail or level of abstraction or granularity of the model. In general, mathematical analysis models can be classified according to the type of model data, parameters and mathematical expressions, or the degree of model *refinement* and *fidelity*. In the remainder of this section we discuss the different types of models in terms of these classifications.

5.4.2. Model classifications based on the nature of model

Analysis models can generally be classified into the following basic categories: qualitative or quantitative, continuous or discrete, deterministic or stochastic, static or dynamic, linear or nonlinear, or any combination of these categories (Figure 5.22). Here we discuss these categories and highlight the main characteristics of the models accordingly.

Figure 5.22:

Analysis models vary based on their mathematical nature.



5.4.2.1 Qualitative and Quantitative Models

Analysis factors can be classified into two distinct types: qualitative, or quantitative. Qualitative analysis is subjective while quantitative analysis is objective. Knowing the difference between these two types is crucial in understanding the techniques for evaluating performance and behavior characteristics.

The qualitative analysis approach involves mostly more words than numbers. Some qualitative methods for this type of analysis include brainstorming, professional experience, questionnaires, interviews and surveys. Qualitative analysis is not physics based but instead is based primarily on subjective opinions, perception and judgment. The difficulty with this type of analysis is that its results cannot be generalized or extended to broader application with the same degree of certainty. Data used for this analysis is not only hard to collect and measure but also creates differences of opinion when interpreted and yields performance assessments that could be conflicting over time.

The quantitative approach on the other hand is mostly physics based, where precise features are identified and enumerated, and models are built to predict artifact or system behavior. Results and findings, if analysis was carried out correctly, are reliable and can provide direct comparisons between different design solutions. The data in this approach is much easier to collect, implement and process using computers. Most of the models used in this thesis are quantitative models.

Although the quantitative approach introduces an easy way by which stakeholders examine and understand design solutions, and can present a quick fix when performance data is required for investment justification purposes, there are some precautions that should be taken into account when dealing with this approach. Results can often lead to simplistic judgments while disregarding the bigger and more complex picture. At the same time, the evaluation process, described here as being reliable, can be distorted if we only pay attention to what is easily measurable and ignore factors that are not.

5.4.2.2 Continuous and Discrete Models

Variables in general are either continuous or discrete. Real numbers for example represent continuous variables. Between any two values of a given continuous variable there always exist an infinite number of other possible values consisting of intervals. These continuous variables can be represented by functions. Variables within these functions are continuous themselves. On the other hand, there are variables that are clearly distinct from each other, and are called discrete variables where the set of possible values consist of only isolated points. Examples of these variables include integers.

If all the data, parameters and relationships associated with a mathematical analysis model are continuous, this model is said to be continuous. It is otherwise said to be discrete (Jacoby and Kowalik, 1980). Discrete models are, however, not always used to model discrete systems, and vice versa (Averill, 2006). Deciding when to use discrete or continuous models for any given systems relies basically on the specific objectives of the study.

Models are not necessarily continuous at all times; they can be continuous only at certain time instants. If the interactions between the variables of these models occur at discrete times only or are separated by intervals where no interaction happens, these models become discontinuous, such as the case with models that involve stochastic effects.

5.4.2.3 Deterministic and Stochastic Models

Mathematical models are deterministic if elements within them are so specified and do not contain any probabilistic or random components to the extent that their behavior, performance or operation can be uniquely determined. As soon as model relationships and input quantities are specified, the output of these models becomes determined with no uncertainties involved (Averill,

2006; Maki and Thompson, 2006). Therefore they perform in the same manner for a given set of initial conditions.

Models are said to be stochastic if they involve uncertainties or stochastic data or elements, or if the parameter values are determined in terms of probability distributions and random input components rather than unique values. Stochastic simulation models, also known as Monte Carlo simulations, employ random number generators for modeling random events. The output result of these models is a probabilistic or random model behavior, performance or operation, therefore they must be seen as approximations or estimates of the actual model characteristics (Jacoby and Kowalik, 1980; Averill, 2006).

A stochastic model thus makes predictions about the probabilities of events and expected values of numerical outcomes (Maki and Thompson, 2006). These predictions intrinsically involve uncertainty regardless of how much is known about a specific situation. This is not the case with deterministic models where predictions are made in specified terms and involve no uncertainties. The reality is, however, that the mathematical description of many models in the real world, and especially in social sciences, involves uncertainty and chance to a great extent.

Selecting and deciding on the type of model to be used, whether deterministic or stochastic, involves a number of factors, but is mainly left to the model developer. Sometimes both deterministic and stochastic models are used for the same situation, augmented by a validity check through comparing predictions from types of models. Deterministic models are regularly used as first approximations in cases where stochastic models seem more suitable for the situation under examination but prove too complex. Generally however, the pros and cons of each type of model vary from one situation to another, and the predictions of each type are not necessarily better or worse than the other for all situations (Maki and Thompson, 2006).

5.4.2.4 Static and Dynamic Models

Mathematical models are considered dynamic or unsteady if their behavior is variable with time. Time here thus exists in the model as an independent variable. Dynamic simulations typically model the changes that take place in a system in response to variable input signals. Differential equations are thus used to represent unsteady-state models.

Static or steady-state models, however, are characterized by constant behavior that is not variable with time. They represent a system at a specific instant in time, and are therefore used to represent systems in which time plays no significant role (Averill, 2006). Steady-state models use equations that define the different relationships between the modeled system elements. In doing so, they tend to identify a state of equilibrium for the system. These models are sometimes employed to simulate physical systems to provide a simpler modeling case before any dynamic simulation is done.

There are other cases where an unsteady prototype behavior is represented by a sequence of steady-state modeling experiments. This yields models known as quasi-steady-state models (Jacoby and Kowalik, 1980).

In some cases, unsteady models can be used for studying the behavior of a steady-state prototype or artifact. The reason for this is mostly computational. A frequently used strategy for solving steady-state modeling problems employs solutions of unsteady-state problems in an attempt to approach the desired steady state. This occurs through a process known as recursive modeling (Jacoby and Kowalik, 1980), where an unsteady or quasi-steady-state model is said to be recursive if its state at a specific time instant or interval is dependent on its state at an earlier time instant or interval. Recursive modeling is thus typically related to time as the model state at a certain time is not defined without defining it at an earlier time.

5.4.2.5 Linear and Nonlinear Models

Typically in any mathematical model, variables are acted upon by a number of operators that include algebraic operators, functions, differential operators, etc. A mathematical model is said to be linear if all these operators introduce linearity. It is considered nonlinear otherwise. This nonlinearity is mostly related to chaos and irreversibility. Generally, but with a few exceptions, it is more difficult to study nonlinear systems and models in comparison to linear models.

5.4.3. Analysis Algorithms

High-fidelity simulations are generally considered better predictors of performance than low fidelity simulations, as they better resemble the artifact if administered correctly. However, the amount of fidelity necessary to guarantee good prediction is unknown. Also, there are some disadvantages to high-fidelity simulations that may render

them less useful, such as trading off speed for accuracy. These types of simulations are not as quick and easy to construct as low-fidelity models.

Low-fidelity prototypes are generally used to quickly demonstrate general artifact performance and abstract conceptual approaches in early design stages without providing much detail or requiring much investment in development. They are mostly used if some required data for the analysis model are not available, if the model cannot be easily quantified, or if a high-fidelity analysis is beyond the scope and accuracy level of the design description. Low-fidelity models also require a facilitator who knows and understands the domain in detail in order to illustrate or test the model.

Based on the degree of fidelity, analysis models can be classified into empirical models, theoretical models, and reduced-order (or approximation) models.

Empirical models, which are typically low-fidelity models, are derived from observation and approximate data fitting rather than physics and first principles.

Theoretical models, on the other hand, are more physics-based and are derived using first-principle equations. They include both analytical and numerical models. Analytical models are mostly low-fidelity models whereas numerical models tend to be high-fidelity (high order) models that include models like Finite Element Analysis (FEA) and Computational Fluid Dynamics (CFD).

Reduced-order or approximation models are surrogate models that provide simplified abstractions and calculations. They include response surface models, neural networks and Kriging models. These models approximate the behavior of a design solution as closely as possible while maintaining low-fidelity, which is computationally cheaper.

Although complexity typically enhances the fit of a model, it sometimes makes the model difficult to understand and operate on, in addition to introducing some computational problems such as numerical instability. Engineers thus often make and accept some approximations and reduce the model size appropriately in order to obtain a more robust and simple model.

The Occam's Razor principle is specifically applicable to modeling. Among models that have almost similar predictive accuracy, the simplest one is the most desirable. During the model selection

process, the designer must choose the best compromise between the demand for simplification and the necessity to clearly identify, describe and rate the targeted physical mechanism. A trade-off must be made between fidelity and analysis time and between simplicity and the accuracy of the model.

In the following sections, our review of analysis algorithms will focus on theoretical models and surrogate models rather than empirical models.

5.4.3.1 Theoretical Models

As mentioned earlier, theoretical models comprise both analytical and numerical models. Techniques in these models tend to explore the behavior of a model. In most cases, however, finding the model in the first place is the most difficult, interesting, and important question (Gershenfeld, 1998).

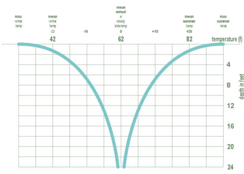
5.4.3.1.1 Analytical Models

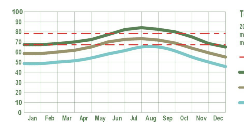
Analytical models are typically employed when the model and the relationships making it up are simple enough such that mathematical methods (e.g. algebra, calculus, or probability theory) can work with these relationships and quantities to obtain precise and explicit information regarding questions of interest (Averill, 2006). This information is known as an analytical solution or a closed-form solution that can be simply arrived at with merely paper and pencil (Gershenfeld, 1998). This analytical solution allows for the prediction of system behavior through a set of initial conditions and parameters (figures 5.23 and 5.24).

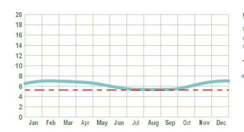
Analytical modeling is mostly done with analytic functions (Saff & Snider, 1993), and therefore the functions encountered are always assumed to be expanded in a power series. Analytical models are still considered very significant due to their power. It is almost always possible to deduce everything that needs to be known about a system using these models. This comes however at the expense of limited applicability, as many systems in the world are too complex to be described in this manner (Gershenfeld, 1998).

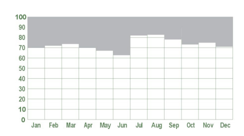
Figure 5.23:

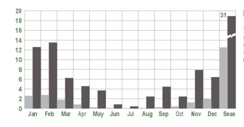
Simple analytical models are used to assess the behavior of a building skin.

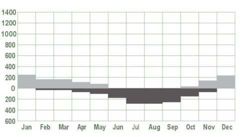


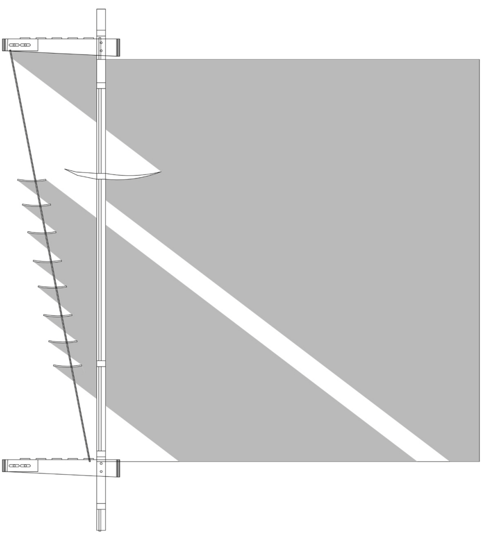


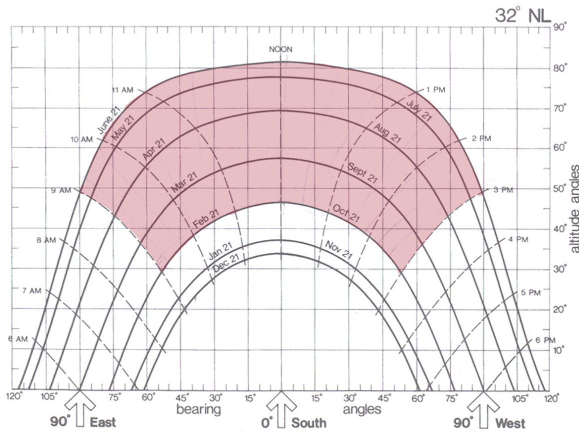


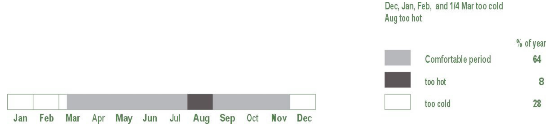


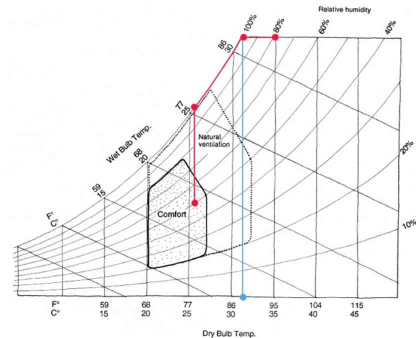


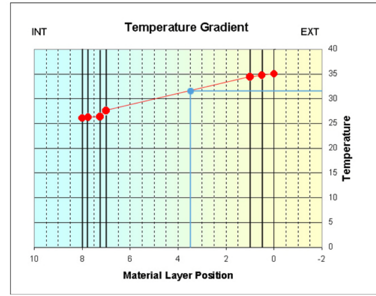












Temperature Gradient Calculation

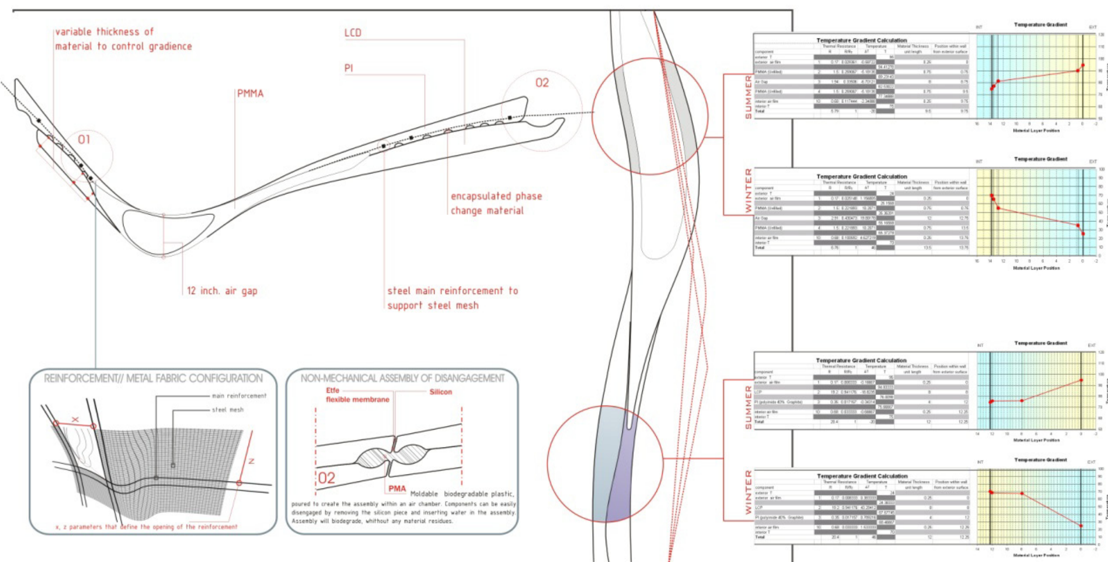
component		Thermal Resistance R	R/R_T	Temperature ΔT	T	Material Thickness unit length	Position within wall from exterior surface
exterior T					35		
exterior air film	1	0.17	0.006905	-0.06214		0.25	0
					34.93786		
Metal Siding 1/2"	2	0.68	0.02762	-0.24858		0.5	0.5
					34.68928		
Plywood 1/2"	3	0.64	0.025995	-0.23396		0.5	1
					34.45532		
M.F. Batt 6"	4	19	0.77173	-6.94557		6	7
					27.50975		
Alum. Sheet Vapor Barrier	5	3	0.121852	-1.09667		0.25	7.25
					26.41308		
Gypsum board, 1/2"	6	0.45	0.018278	-0.1645		0.5	7.75
					26.24858		
interior air film	10	0.68	0.02762	-0.24858		0.25	8
interior T					26		
Total		24.62	1	-9		7.75	8

Figure 5.24:

Simple analytical models are used to assess the behavior of a building skin.

Project Credits: Anas Alfaris Alexandros Tsamis

Using such simplified low-fidelity analysis models has many benefits. First, these models generate a quick and rough estimate of the artifact's performance. Second, it does not require detailed information or information that is not currently available at this early stage of design, as opposed to a high-fidelity model. Most importantly, however, these low-fidelity analysis models introduce less computational burden (Huebner et al., 2001). This type of model should be treated cautiously since using simplifying assumptions that ignore problems or difficulties could sometimes leads to erroneous results or inaccuracies.



5.4.3.1.2 Numerical Models

Although I stated above that analytical models are not computationally intensive, analytical solutions can occasionally be complex and call for immense computing resources. Obtaining a numerical solution for a situation where an analytical formula exists in theory, could be a very difficult task, such as the example of inverting a large nonsparse matrix (Averill, 2006).

Also, in many cases the closed form solution implied by analytical modeling is not usable for modeling experiments (Jacoby and Kowalik, 1980). If for example the model consists of a series that comprises many terms that need to be computed for solution accuracy purposes, the process of reformulating the problem as a numerical problem might be more economical. This reformulation

takes the form of a sequence of consecutive approximations to the solution that are computed in an iterative manner. In these iterations, each approximation is "better" than its predecessor.

However, it is always preferred to study a mathematical model analytically rather than numerically if an analytical solution exists and is computationally efficient. Analytical models still remain an important factor in some approximation techniques using computers. This extends to include numerical methods, where these methods can use bits and pieces of analytical solutions to render the numerical steps more effective. They also employ symbolic methods that can extend quantitative abilities and introduce important qualitative implications, such as in enhancing the methods to perform higher-order approximation theory (Gershenfeld, 1998).

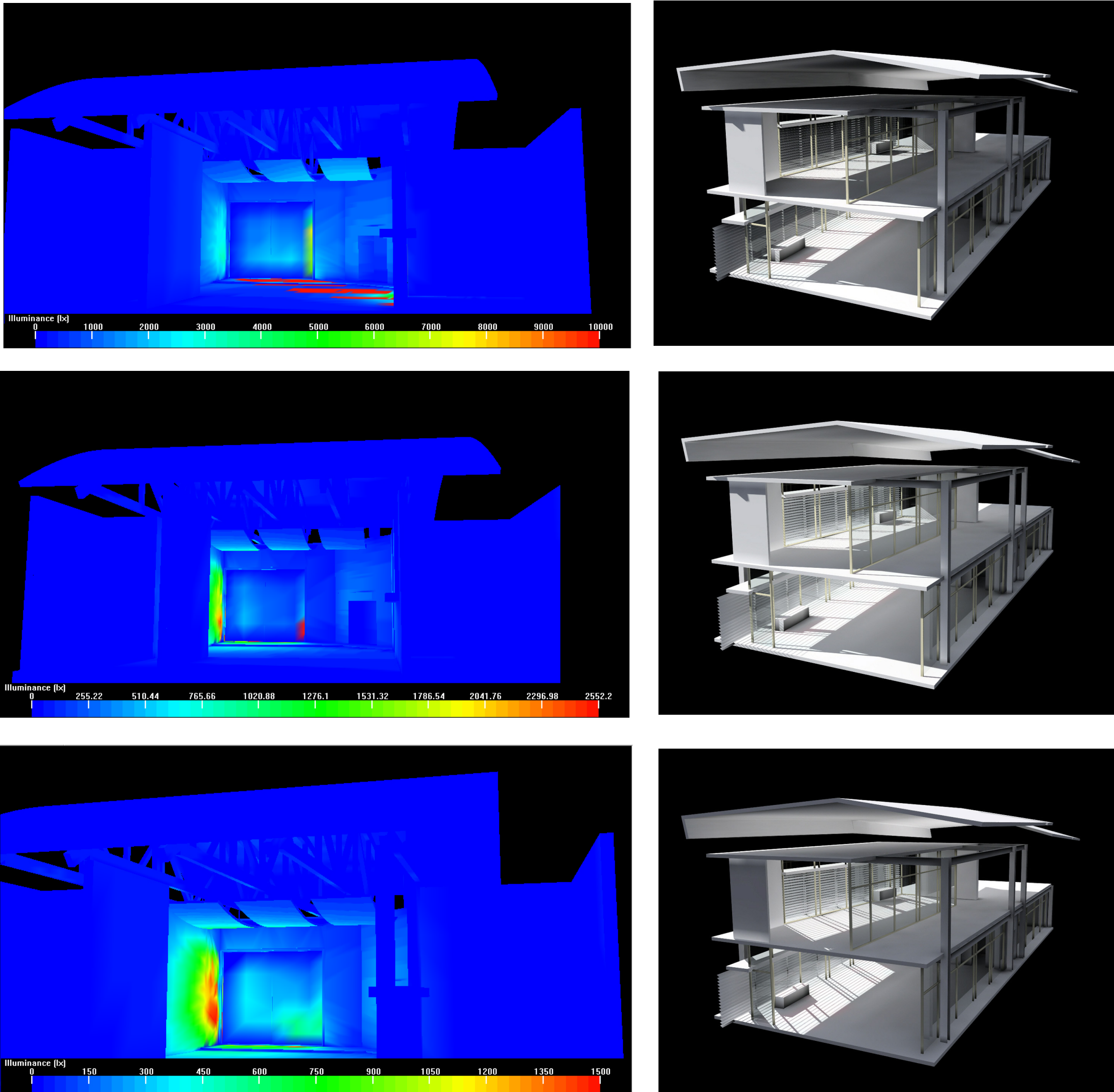
Many real-world systems are too complex for analytical modeling or evaluation. It is clear then that not many differential equations can really be solved with precisely the same effort analytically. As we move farther from linearity, it becomes obvious that special techniques are required and enormous effort must be made to be able to write down a closed form or analytical solution (Gershenfeld, 1998). In the computer environment, differential equation models are typically reformulated and expressed in terms of difference equation approximations. Therefore the issue is reduced computationally to solving problems in the form of a set of algebraic equations (Jacoby and Kowalik, 1980).

Models should thus be studied by means of simulations. This is done by numerically evaluating the model inputs to observe their effect on the output performance measures (figure 5.25). In this evaluation, an abundance of data is collected to provide estimates of the required real model characteristics (Averill, 2006). The main goals of modern numerical analysis are the design and analysis of techniques that aim at obtaining approximate solutions to complex problems rather than getting exact answers, while at the same time preserving reasonable bounds on errors, since it is impossible to get exact answers in practice.

According to Jacoby and Kowalik (1980), numerical solution methods for modeling problems basically have three main goals. These include computational efficiency, precision and error control, and solution convergence, where it is possible at some point to end the computation. They also add a few more points that could also be taken into account, such as the ability to solve altered or extended formulations, the availability of software that can successfully implement a specific method, and the conceptual clearness and ease of use of solution methods.

Figure 5.25:

A high-fidelity analysis model for day-lighting is used to assess the lighting quality in different spaces.



Several efficiency measures have been developed for numerical methods. These include the counts of arithmetic operations required to solve problems and the order of solution convergence in the case of iterative processes. One of the most practical measures of efficiency, however, is the total computer resource required for solving the given problem, including time and storage space (Jacoby and Kowalik, 1980).

It is often hard to relate the latter measure to more theoretical properties of the model formulation and solution method. There are thus a group of mathematical techniques that help reduce the total computer resource needed for a solution. These include the decomposition of large-scale problems into smaller components in a semi-independent manner, using linearization if possible, data compression, and the process of reducing problems with unknown degrees of difficulty into well-known problems with which the model user is familiar and has relevant experience.

Accuracy of the numerical solution is another issue, which largely relies on the quality of data, the degree of model approximation, and the numerical properties of the solution method. Any one of these factors can nullify the solution results by itself. Recent developments in numerical analysis have made it easier for model users to comprehend many of the issues regarding error analysis and the conditioning of numerical problems and algorithms. These developments have enabled the understanding of the conditions under which any analysis mathematical model can be successful and useful.

The numerical results of the model should be interpretable and validated in the system space or else this information will remain uninterpretable in the analysis model space and would thus become unfamiliar to the model user. The mathematical modeling problem itself must be solvable in order to conduct the experiments correctly. This implies two conditions, existence and stability, meaning that the solution to the problem must exist theoretically and at the same time must always rely on the given side conditions. Any discontinuity must be accounted for appropriately.

Another condition for the success of the analysis model is uniqueness, where it should be understood beforehand whether the mathematical modeling problem allows for more than one solution or not. Whether or not the problem is well-conditioned is another important factor. This implies knowing whether small approximations in problem data result in small approximations in the final solution or not.

Finally, the feasibility of a computational process for numerical approximation is another important condition, where the model user must be allowed to keep the approximation error under control (Jacoby and Kowalik, 1980). Studying errors constitutes a very significant part of numerical analysis.

Errors can be introduced in the solution of the problem in a variety of ways. Round-off errors exist because it is impossible to represent all real numbers precisely on finite-state machines such as digital computers. Truncation errors exist after an iterative method is terminated and the approximate solution turns out to be different from the exact solution. Discretization errors also occur in the same manner, when the solution of the discrete problem does not match the solution of the continuous problem. As a general rule, an error generally propagates through the calculation once it is generated. If this propagation does not grow and accumulate in the input data and intermediate calculations causing a meaningless output, the algorithm is said to be numerically stable. This stability occurs only if the problem is well-conditioned, implying that the solution only changes by a small amount when the problem data is changed by a small amount. A well-conditioned problem does not necessarily guarantee the numerical stability of the algorithm, but an ill-conditioned problem definitely leads to error accumulation and consequently instability.

There have been several methods and algorithms developed for numerical models. These could be direct or iterative methods. Iterative methods are generally more common in numerical analysis than direct methods.

In direct methods, the solution to a given problem is computed in a finite number of steps. The accurate answer can be provided through these methods if they are performed in infinite precision arithmetic. Finite precision is used in practice, and the end result represents an approximation of the true solution assuming stability. Examples of these methods include Gaussian elimination, the QR factorization method for solving systems of linear equations, as well as Cholesky and LU factorization (Trefethen and Bau, 1997).

Iterative methods are usually needed for large problems in computational matrix algebra. Unlike direct methods, iterative methods are not expected to be complete in a specific number of steps. They start from an initial guess to construct consecutive approximations that converge to the exact solution only in the limit. To determine when an accurate solution is found, a convergence criterion is specified. In general, even if iterative methods use infinite

precision arithmetic, the solution would not be reached within a finite number of steps. Some examples include Newton's method, the bisection method, and Jacobi iteration (Trefethen and Bau, 1997).

Some methods, although direct in principle, are used as if they were not, such as GMRES and the conjugate gradient method. In these methods, the required number of steps to obtain an exact solution is large to the extent that approximations are accepted similar to the case of iterative methods.

Many methods have been developed for solving systems of linear equations. Standard direct methods that use matrix decomposition include Gaussian elimination, LU decomposition, Cholesky decomposition for symmetric and positive-definite matrix, and QR decomposition for non-square matrices. For large systems, iterative methods are preferred, such as the Jacobi method, Gauss–Seidel method, the successive over-relaxation and conjugate gradient method (Trefethen and Bau, 1997). Root-finding algorithms and linearization are both techniques that are used for solving nonlinear equations. Newton's method is also used but when the function is differentiable and the derivative is known.

There are many other methods that are used to solve partial differential equations. Discretization is an approach that describes the process in which a continuous problem is substituted by a discrete problem whose solution is known to approximate that of the continuous problem. These methods face a major challenge that requires generating an equation that approximates the equation to be studied while being numerically stable. One of the methods used in this regard is the Finite Element Method, which is a good choice for solving partial differential equations. We describe this method in the following section, followed by a brief overview of computational fluid dynamics as another prominent numerical analysis method.

The Finite Element Method (FEM)

It is becoming more and more important in engineering situations to provide approximate numerical solutions to problems instead of closed-form or analytical solutions. These solutions rarely exist, for example, in cases where the geometry or any other feature of the problem is irregular or arbitrary (figure 5.27).

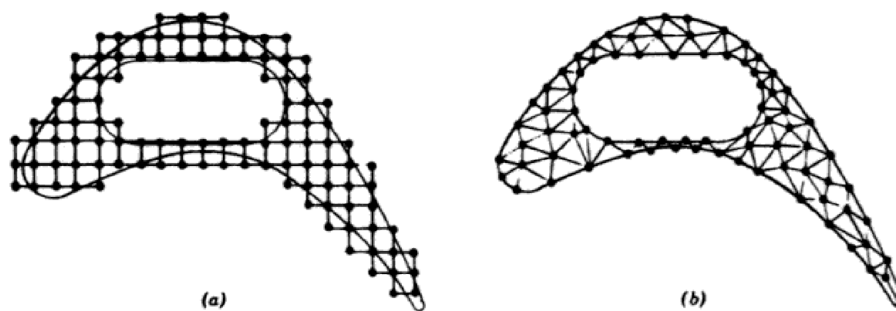
One of the common approximate numerical analysis methods is the finite difference scheme. The finite difference model of a problem provides a pointwise approximation to the governing equations. As

more points are used, the model is improved, as it employs difference equations for an array of grid points. Although finite difference techniques can solve fairly difficult problems, they are harder to use with irregular geometry or unusual specification of boundary conditions.

Another approximate numerical analysis method is the Finite Element Method (FEM). As opposed to visualizing the solution region as an array of grid points, FEM visualizes it in terms of numerous small and interconnected sub-regions or elements, and so provides a piecewise approximation to the governing equations (Huebner et al., 2001). Figure 5.26 shows how a finite difference model and a finite element model can be used to represent a complex geometrical shape. Using the finite difference techniques, a uniform mesh would cover the whole solution region, but the boundaries would have to be approximated through horizontal and vertical lines analogous to stair steps. The FEM would however provide a better approximation to the region using triangles as the simplest 2D elements in addition to straight lines of different inclinations that also give better approximation to the curved boundary shape. The main purpose here is to show that the FEM is better suited than the finite different method for problems that comprise complex geometries and not necessarily for all types of problems.

Figure 5.26:

(a) Finite difference and (b) finite element discretizations of a turbine blade profile (Huebner et al., 2001).



FEM first evolved in civil and aeronautical engineering where there was a need to solve complex problems related to elasticity and structural analysis. It was later extended and applied to continuum mechanics and a wide range of engineering problems (Huebner et al., 2001).

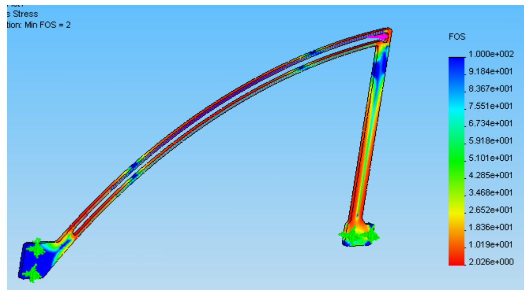
The FEM represents a numerical technique for finding approximate solutions of partial differential equations in addition to integral equations. This is done by either eliminating the differential equation completely or turning it into an approximating system of ordinary differential equations that are then integrated numerically by standard techniques, such as Euler's method or Runge-Kutta.

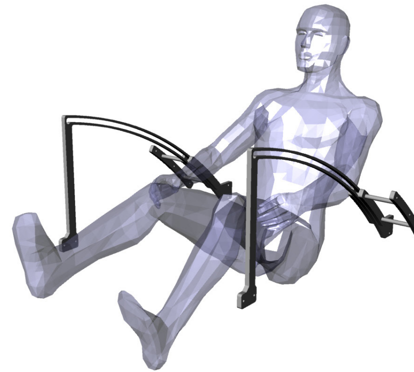
Briefly, the way by which FEM works involves primarily the concept of meshing. This is done after model geometry is developed in a CAD program, and the problem is identified through defining material properties and boundary conditions. Meshing the model basically deals with defining a finite number of elements to represent the geometric structure or solution region (Huebner et al., 2001). These elements can represent very complex shapes since they can be assembled in various ways. Although more elements imply higher accuracy, they also entail more computational time for reaching a solution. A sense of balance thus must be maintained between the required time to solve a problem and the acceptable level of error that can result from high complexity.

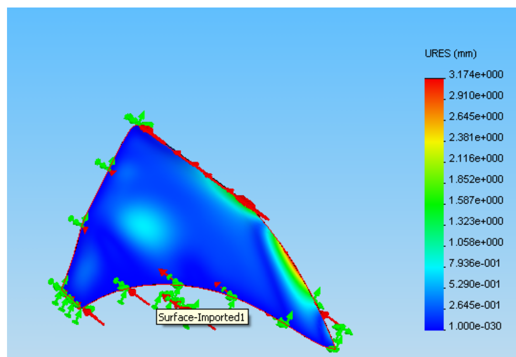
This finite element discretization transforms the problem from a problem of infinite unknowns into one of a finite number of unknowns. After dividing the solution region into elements, the unknown field variables are described in terms of assumed approximating functions within each element known as interpolation functions.

The points at which these functions are defined in terms of the field variable values are known as nodes or nodal points. There are boundary nodes on the element boundaries where adjacent elements are connected in addition to some interior nodes. The behavior of the field variable is entirely defined by its nodal values and also by the interpolation functions for the elements. As the nodal values here become the unknowns themselves, as soon as they are found, the interpolation functions define the field variable during the process of putting the elements together.

This shows that the chosen interpolation functions greatly affect the nature of the solution and the degree of approximation, which does not rely solely on the size and number of elements. These functions are not selected arbitrarily, as some compatibility conditions need to be fulfilled. They are rather chosen mostly so that the field variable or its derivatives are continuous across adjoining boundaries of elements. This is then applied to the formulation of different element types.







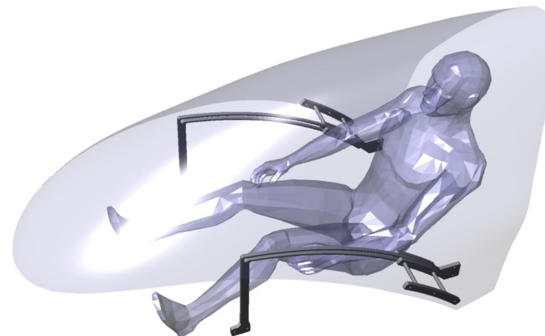


Figure 5.27:

Finite Element Analysis of different components in this vehicle egress and digress system.

Project credits: Anas Alfaris, Nii Armar and Martin McBrien

Computational Fluid Dynamics (CFD)

Computational fluid dynamics (CFD) was initiated in the early 1970's as a branch of fluid mechanics that roots from physics, numerical mathematics and computer sciences, and uses numerical methods and algorithms to solve, analyze and simulate problems related to fluid flows (Blazek, 2001). CFD methodologies today are regularly used in aircraft, car, ship, and building design (figure 5.28).

One of the first applications of CFD was the simulation of transonic flows based on solving the non-linear potential equation. In the early 1980's, the solution to the first 2D and later 3D Euler equations became possible. It was then also possible to compute inviscid flows in aircraft configurations owing to the increasing speed of supercomputers and many numerical acceleration techniques such as multigrid (Blazek, 2001).

There was a shift of focus in the mid 1980's to the more important and demanding simulation of viscous flows which was governed by

the Navier-Stokes equations. In addition, a number of turbulence models were developed. (Blazek, 2001).

The rising demand on the complexity and fidelity of flow simulations led to further development and greater sophistication of grid generation methods. These methods evolved from simple structured meshes using algebraic methods or partial differential equations in the decomposition of grids into topologically simpler blocks, known as the multi-block approach. Subsequent research focused, however, on developing unstructured grid generators due to the slow nature of the structured multiblock grid in dealing with complicated geometry. These unstructured generators or "flow solvers" yielded considerably reduced setup times with minor user intervention in addition to introducing solution based grid adaptation (Blazek, 2001).

Before any numerical solution method can be implemented, the way by which the method affects the stability and convergence behavior of the CFD code must be determined or at least approximated. Blazek (2001) refers to the Von Neumann stability analysis as providing a good assessment of the properties of a numerical scheme in this context.

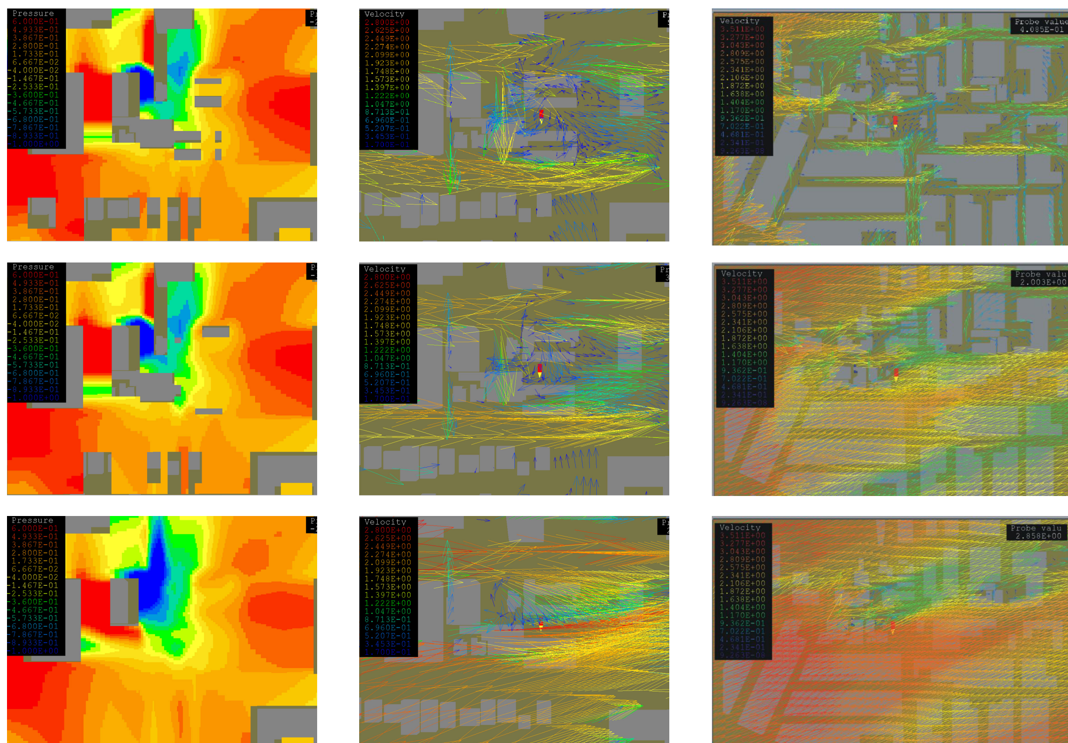


Figure 5.28:

CFD Model for a building site to study the airflow around the building.

Project Credits: Anas Alfaris, Kenneth Namkung Meredith Elbaum

Since CFD methods are primarily concerned with solving equations of motion of fluids and the interaction of fluids with solid bodies, the governing equations are Navier-Stokes equations that describe the motion of viscous fluids and Euler equations that describe the motion of inviscid fluids (Blazek, 2001). Through a series of simplifications, these equations can yield linearized potential equations. By first removing certain terms that describe viscosity, Navier-Stokes equations can be simplified to generate Euler equations. By removing terms that describe vorticity, full potential equations can be produced.

In CFD, the geometry or physical bounds of the problem are defined during preprocessing. One of the challenges is generating structured or unstructured body-fitted grids around complex geometries which would then be used to discretize the governing equations in space. In this process, the volume that the fluid occupies is divided into discrete cells. These cells constitute a mesh which is either regular or irregular in form. The quality of the grid greatly affects the precision of the flow solution (Blazek, 2001). In terms of computational cost and memory storage, the regularity or irregularity of the formed meshes is highly significant.

A discretization error is introduced for each discretization of the governing equations. The discretization scheme thus has to satisfy multiple consistency requirements to guarantee that the solution of the discretized equations closely approximates that of the original equations.

Following discretization, physical modeling is then defined, including for instance equations of motions, enthalpy, and radiation. This is followed by defining boundary conditions, where the fluid behavior and properties at the problem boundaries are identified in order to account for the specific features of a particular problem and find a unique solution for the governing equations. There are two types of boundary conditions: physical and numerical (Blazek, 2001). Initial conditions are also identified in case of transient problems.

As the simulation process begins, equations are solved iteratively as a steady-state or transient. Suitable algorithms are applied to solve the equations of motion, both the Euler equations for inviscid and the Navier-Stokes equations for viscous flow. Most often other equations are solved in parallel to the Navier-Stokes equations. These can include equations describing mass transfer, chemical reactions, heat transfer, etc.

In this process of solving the equations, either one of two approaches can be used. The steady-state governing equations can

be solved directly, or the unsteady governing equations can be integrated with respect to time. The latter approach is also known as time-stepping schemes (Blazek, 2001). Time-stepping can be divided into two classes, one that comprises explicit time-stepping schemes, while the other class comprises implicit time-stepping schemes. After solving the equations, the final process in CFD is the analysis and visualization of the resulting solution.

5.4.3.2 Approximation Techniques

As mentioned earlier, most engineering design problems require experiments or simulations in order to evaluate design objectives and constraint functions as a function of design variables. As single simulations are time consuming, routine tasks such as design optimization, design space exploration, sensitivity analysis and what-if analysis become impossible to achieve as they require numerous simulation evaluations.

Since there is no novel architecture that is expected to address these issues and offer a direct solution to this problem, it is crucial to recognize the importance of developing approximate analyses that are quicker, simpler and more efficient in runtime (Kroo, 1997a). Approximation concepts are primarily used to create surrogate behavior models that replace expensive computer analysis and simulation programs for the purpose of quick analysis predictions. A wide variety of techniques has been developed for generating surrogate models, ranging from classical forms of curve fitting to more recent concepts of neural networks and Kriging (Papalambros and Wilde, 2000).

Surrogate models extract simpler analysis models from sophisticated ones by applying various data-handling techniques. The actual internal simulation code of the sophisticated model and how it works is not assumed to be known or even comprehended. Using the sophisticated model as a source of "computational experiments", analogous to physical experiments, provide a collection of data points (Papalambros and Wilde, 2000). Building on these data points, a surrogate model can be derived with new simpler functions that represent the system functions in an explicit manner with reasonable accuracy.

These surrogate models, which now contain the simpler functions, then replace the sophisticated model in any succeeding uses. This dramatically minimizes computational load in large and complex design models that comprise multiple analysis models. The sophisticated model can be used after the final design is attained in

order to obtain more accurate estimates (Papalambros and Wilde, 2000).

Using these simpler models facilitates the computational use of various design exploration techniques, including optimization and expensive probabilistic analysis/optimization methods in particular, in large-scale complex design problems (Koch et al., 2002).

Not only do these surrogate models increase efficiency, but they also get rid of the computational noise in simulation programs comprising outputs that fluctuate frequently upon gradual changes in input parameters (Phoenix Integration, 2004), such as that in large-scale problems, eigen frequency problems, impact problems, and nonlinear problems (Sakata et al., 2003). As this noise typically has an unfavorable effect on optimization by generating many local optima, approximation models can thus smooth the response functions and increasingly enhance convergence.

Classical curve fitting techniques

The basic concept behind curve-fitting techniques involves collecting data from experiments or other empirical sources to derive functional relationships between dependent and independent variables. This data, which is typically a function representation in tabular or graphical form, is converted into more convenient and useful equation or algebraic form (Papalambros and Wilde, 2000). Through understanding the relationships between these variables, models could be developed for prediction of behavior (Koch et al., 2002). In general, this approach is beneficial for further analysis or for computational purposes.

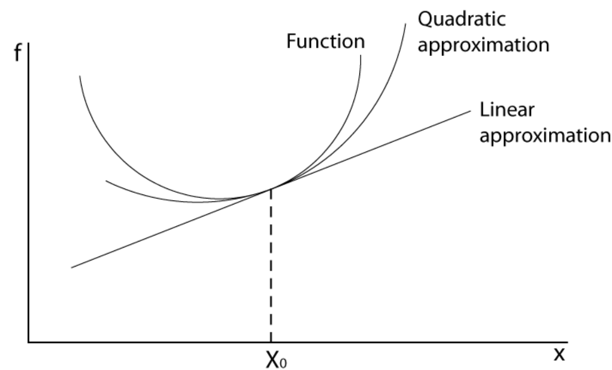
One of the useful approximation tools in this context is the Taylor-series approximation, which is based on modeling functions as truncated Taylor series with typically only first-order terms included. Unknown functions are therefore modeled as first-degree polynomials where the constant term is defined by the function value at the baseline design. The coefficients in the polynomial are gradients of the function that are usually defined using the finite difference method (Koch et al., 2002).

The Taylor series representation is, however, different from a general polynomial. The basic distinction is that the Taylor series uses localized derivative information (figure 5.29) while polynomial fitting uses information from different points (figure 5.30). If the derivatives are calculated numerically with finite differences, these differences begin to disappear. This is typical of all numerical schemes that

involve derivatives. It is worth noting that functional forms, other than polynomials, may be used for curve fitting (Papalambros and Wilde, 2000).

Figure 5.29:

Taylor series uses localized derivative information at point *Xo.*



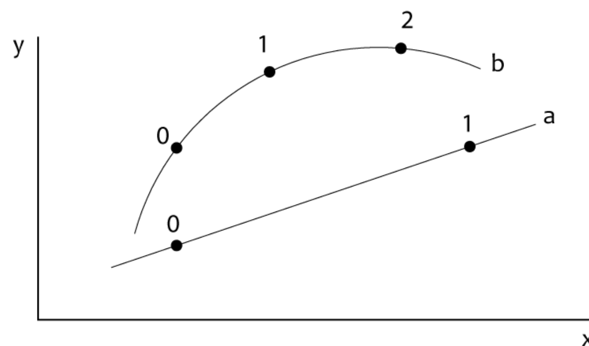
There can be conditions where the polynomial with the least total deviation from the data points simply does not pass through any of the points. The best-fit polynomial has to be defined then through some sort of formal procedure.

One way to do this is calculating absolute values for the deviations from the data points and locating the polynomial that minimizes the sum of those values.

Another way, known as the least-squares fit method (Papalambros and Wilde, 2000), finds the polynomial that minimizes the sum of the squares of the deviations. This method can be implemented to not only polynomials, but also to any function chosen for data representation.

Figure 5.30:

Polynomial fitting uses information from different points for each curve.



There are some difficulties to working with least-squares fit methods, however. Basically employing this method does not necessarily imply that the curve fitting is good. If the data was scattered for instance, the method could produce a correct answer,

but one that is meaningless.

Another faulty result can be generated especially if the degree of the fitting curve is too low, where the method could produce the best possible “bad” solution.

Overfitting can also take place, where a function form seems too “wiggly” between data points and has to be followed closely (Papalambros and Wilde, 2000).

Another problem arises from ill-conditioning. This results from increasing the polynomial degree to achieve fidelity which produces a different order of magnitude between low and high-degree terms.

Response Surface Models

Response surface models (RSM), which are mostly low order polynomial models, constitute one of the most common types of approximating methods (Myers and Montgomery 1995).

The number of data points required to fit a RSM is directly related to the number of terms in the model. For any n number of terms, there exists a minimum number of n+1 points, where the additional point is used for estimating the mean or constant term (Koch et al., 2002).

RSM usually requires assuming the order of the approximated base function as the approximation process is conducted via the least square method for the unknown coefficients of the function (Sakata et al., 2003; Kaymaz, 2005). The designer must thus assess the schematic shape of the objective function over the whole solution region.

As this demands a clear perception of the qualitative tendency of the whole design space, it is often difficult to do. So in the case of noisy known objective functions for instance, it is quite adequate to provide a subjectively assumed base function.

One of the other difficulties in RSM that Shi et al. describes is the complexity of applying them based on experimental programming to design problems with a large number of design variables. Yet another difficulty with the response surface optimization methods is their limited range of application (Balabanov and Venter, 2004).

Therefore, simple approximate models, such as response surfaces, can be seen as appropriate only in a constrained region of the design space (Kroo, 1997b).

Neural Network

As mentioned earlier, least-square models usually determine coefficient values within an algebraic expression. But prior to defining those values, the desired form of this expression must be clearly identified. Artificial neural networks (also known as neural networks or neural nets [NN] for short) address this issue through an automated approach for data fitting.

Neural networks represent mathematical or computational models that are based primarily on biological neural networks. They are analogous to the biological neural networks in the sense that functions within the network are conducted simultaneously and collectively by the units without a sharply defined task assignment for each unit.

The core of neural networks is quite similar to that of nonlinear-least squares (Papalambros and Wilde, 2000). They look for a set of coefficients within an algebraic function that reduces the sum of approximation errors of the function evaluation to the minimum at the sample points when compared with data in order to increase the precision of the approximated value at a given sample point (Sakata et al., 2003).

Although neural networks cannot be described as adaptive as such, they can modify their structure according to the flow of external or internal information through the network during the learning phase. Neurons in neural networks behave similarly to power terms in least-square fits, where they constitute the building blocks of the network. Each neuron has a weight *w* and bias *b* that define how the neuron behaves for a given input. It then combines the weighted inputs linearly, adds a bias, and produces an output. The output is between 0 and 1 based on a weighted sum of the inputs. If the sum is greater than a certain *bias* (*b*), the output is considered to be "on" if greater than ½ and "off" if less than 1/2 (Papalambros and Wilde, 2000).

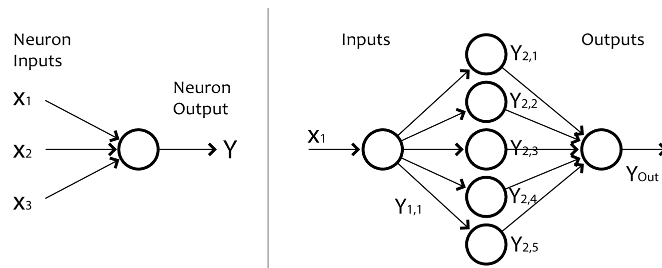
Figure 5.31 shows how inputs to some neurons are considered as outputs from others in complex functions, all connected together to form a neural net. In this net, each neuron has an output that relies only on its own inputs, and so one equation exists for each single neuron. As illustrated in the figure, there is a middle layer of neurons that generates the intermediate values. This layer is known as the hidden layer. The net here can thus be described as a neural net with a single hidden layer consisting of five nodes (Papalambros and Wilde, 2000).

The weights and biases mentioned earlier define the response *y* out for a given stimuli *x* in a similar fashion to the role of coefficients in nonlinear least squares, and therefore should be determined for a specific dataset early on in a process known as training the neural net (Papalambros and Wilde, 2000).

Due to their extremely nonlinear nature however, weights and biases are not unique, implying that any two neural nets with totally different weights and biases can model a specific dataset equally well.

Figure 5.31:

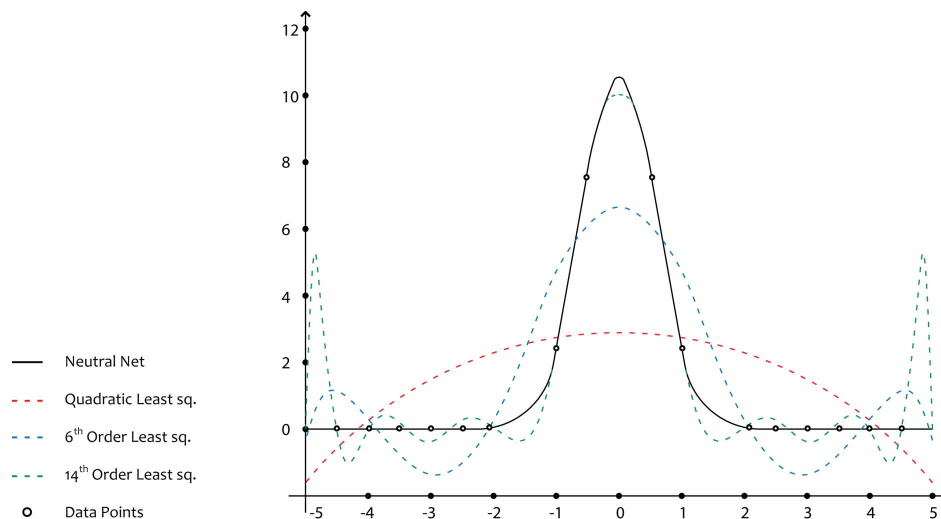
A diagram for a single neuron on the left and a neural net on the right (Papalambros and Wilde, 2000).



It was reported by Carpenter et al. that neural net approximation provides more flexibility to enable fitting than RSM (Balabanov and Venter, 2004). In figure 5.32, a comparison is done between the responses of a neural net and four hidden nodes to least-squares polynomials of different orders (Papalambros and Wilde, 2000).

Figure 5.32:

A neural net and three other polynomials modeling the same data (Papalambros and Wilde, 2000).



Neural networks present a fast and effective method for modeling nonlinear data, and in particular when no algebraic form is readily available. However, they introduce some practical difficulties. Some of these are related to the computational cost accompanying the learning process. Others have to do with requiring a skilled and experienced operator in using neural networks.

Kriging Method

The Kriging method, named after D. G. Krige who first developed it (Papalambros and Wilde, 2000), can also be generally thought of as a mathematical "curve fit" through a set of data generated by the analysis code (Phoenix Integration, 2004). It is a method of spatial prediction or estimation based on minimizing the mean error of the weighting sum of the sampling values (Sakata et a., 2003).

In order to generate the surrogate model for this method, which interpolates Kriging models, the analysis model is executed in a series of runs and the results of each run are stored. The input variable values for this series of runs are selected to estimate the design space in an efficient manner.

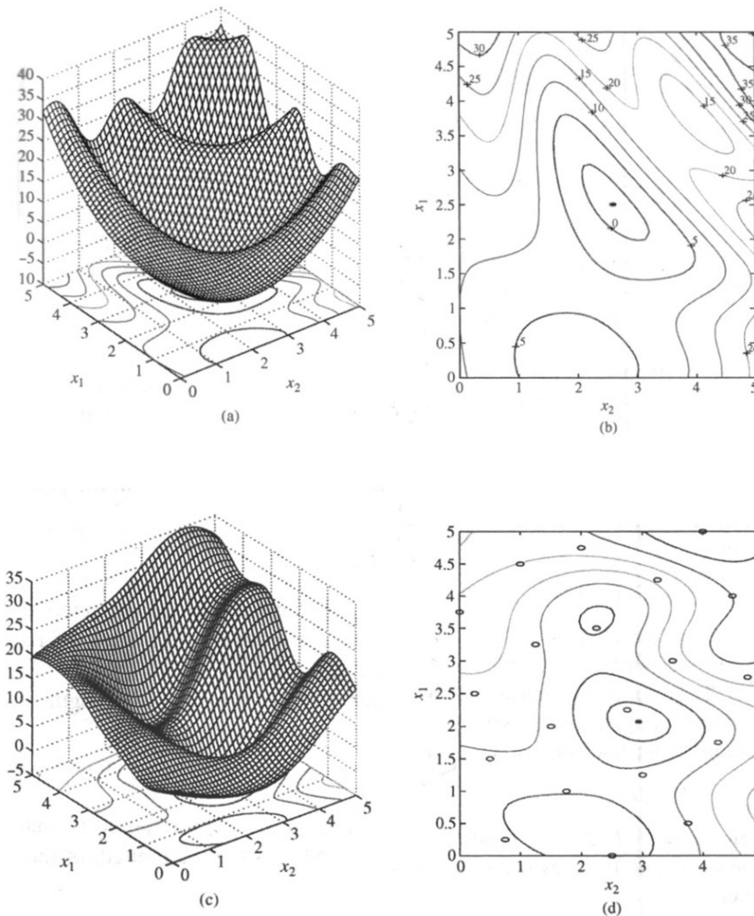
Spatial estimation using the Kriging method involves a number of steps like determining a semivariogram model, estimating parameters for a semivariogram and calculating the weighting coefficients for a spatial predictor. A semivariogram is a variance function in a probabilistic field that expresses data dispersion.

While the Kriging method reduces estimation errors via the variance of estimated values, it demands the assumption of the kind of semivariogram model for the purpose of generating an estimated surface. There are many types of semivariograms including linear models, exponential models, Gaussian-type semivariogram models and others (Sakata et al., 2003). An estimated surface using the Gaussian-type semivariogram model for example should be appropriate for optimizing as it is smooth and continuous.

In general, the Kriging method is preferred over response surface models that are based on experimental programming methods, as well as neural network approximation methods that encounter high computational learning cost (figure 5.33).

Figure 5.33:

A kriging model for a two dimensional function. Two top plots are the actual function and the two lower plots are the kringing model (Papalambros and Wilde, 2000).



Variable complexity approximation (Multi-fidelity Analysis)

Variable complexity approximation models or multiple-fidelity analysis models combine high and low-fidelity analyses to reduce computational cost (Koch et al., 2002). They are generated by means of two analysis tools that model the same physical phenomenon using different degrees of fidelity; one is a high-fidelity and more precise cost simulation code, and the other is a less precise simulation code which is computationally cheaper (figure 5.34).

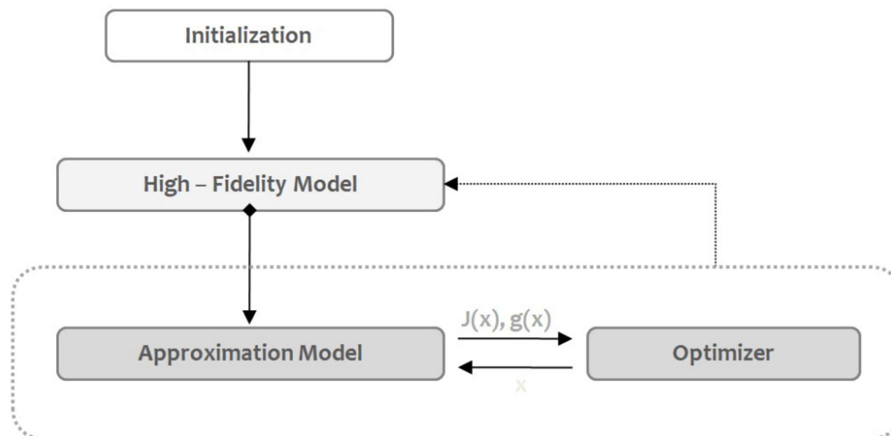
The fundamental idea of these models is using multiplicative or additive correction factors. These correction factors are applied to outputs of lower fidelity efficient codes and used to predict function values that would be obtained if higher fidelity complex codes were used. The factors are acquired by collecting data points using both codes (Koch et al., 2002).

For instance, a response surface can be generated from a small number of high-fidelity analyses. Low-fidelity analyses are then conducted for the same points and a response surface for low-fidelity analyses is thus created. By using the correction factor for the response surfaces or the analysis results, the low-fidelity analysis results can be transformed into high-fidelity analysis results (Balabanov et al., 2004).

An important drawback of this method, however, is that the results of high and low-fidelity analyses have to be correlated from time to time during optimization. This correlation can be complicated in situations involving a large number of design variables and responses, especially when each response uses its own correction factor. This would introduce the limitation more clearly in the number of design variables and responses employed.

Figure 5.34:

Within an optimization both high-fidelity and low-fidelity models can be utilized.



5.5 Evaluation Models

5.5.1 What is an Evaluation Model?

Evaluation models are in essence decision-making tools. An evaluation model provides a quantitative assessment of the effects of design decisions on the system being considered. An evaluation model provides an objective evaluation as opposed to a subjective evaluation of system behavior.

In single objective design problems, the optimization and search direction can be well defined and a single solution, if it exists, could be found. However, as the design develops, more than one objective function will often be identified. These objectives may be competing and therefore trade-offs must be made. This implies that there is no single optimal solution but rather a whole set of possible solutions of equivalent quality (Abraham et al., 2005).

Once there is more than one option, a choice must be made. Rational choice requires a criterion to evaluate the different alternatives and rank them based on a figure of merit describing the quality of a design solution (Papalambros and Wilde, 2000). This should be an objective criterion which will help in the selection of the best solution from the generated alternatives. This criterion for evaluating alternatives is not unique but is rather influenced by many design factors, such as application, timing, the designer's point of view, among other factors. In addition a criterion may change over time as more information is gathered about the design. The formulation of the objective function is vital to the outcome of the design space search.

Addressing multiple objective problems may require techniques that differ from standard single objective optimization methods. Having several objectives leads to a vector objective rather than a scalar one. There are several methods that have been developed to formulate and solve such multi-objective problems.

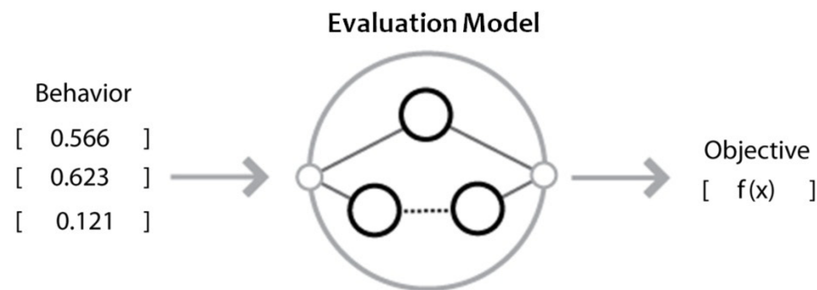
This objective is articulated based on the decision-maker's preferences either before or after the search. When the preference is expressed beforehand, the designer decides how to aggregate different conflicting objectives into a single objective function before the actual search is performed (Horn, 1997). A commonly adopted approach is scalarization. This consists of combining several objectives into one scalar cost function. There are different scalarization methods, such as the weighted-sum approach and the utility function method among others.

When using a single-objective search, the result of the optimization is a single design point (Gries, 2004). If the weights are changed, the full search needs to be repeated. Furthermore, depending on the shape of the objective function that aggregates several objectives, certain regions of the design space might be inaccessible.

When a search is performed before making a decision, the search is performed with multiple objectives at the same time. The solution space becomes partially ordered with a set of optimal trade-offs between the conflicting objectives. This set is called the Pareto optimal set. The actual choice of one of the solutions depends on further constraints or objective functions that apply combinations of the objectives used for the search (Gries, 2004).

Figure 5.35:

Expected input and output of the analysis model.



In the context of the MDDS, Evaluation models help make decisions about the multiobjevte nature of the design problem. If preference is expressed before search then the evaluation model aggregates the behavior of the different analysis models into one single objective that the optimization model can then use to search the design space (figure 5.35). If decision making is delayed after search, then the evaluation model becomes part of the optimization model.

A brief review of multiobjective methods will follow. This review is not intended to be comprehensive, but will focus on the most popular multiobjective methods.

5.5.2 Single and Multi Objective Evaluation and Optimization

When the optimization task is composed of only one objective, the design problem is called a mono- or single-objective problem. The main goal of a single-objective optimization is to find the “best” solution, which implies the minimization or maximization of an objective function that consists of a design vector that includes design variables and is subject to both equality and inequality

constraints. In single objective optimization, the search space is often well defined.

Many design problems need to achieve several objectives such as: maximize performance, minimize deviations from desired levels, and minimize cost. Design problems may also have several multiple conflicting objectives. As soon as there are several -possibly contradicting- objectives to be optimized simultaneously, a single optimal solution no longer exists.

Using a single-objective optimization to solve a problem with several objectives entails grouping all different objectives into one single objective function (Savic, 2002). However, although some design problems may be reduced to a single objective, very often it is difficult to define all the features of the design problem in terms of a single objective. Also a single-objective optimization usually cannot provide a set of alternative solutions that show different objectives against each other. Different objectives may show tight relations to other objectives but optimizing with a single objective may reveal severe trade-offs with respect to the other objectives (Gries, 2004).

Therefore, defining multiple objectives often gives a better idea of the design space and the possible trade-offs between conflicting objectives. The interaction among different objectives gives rise to a set of compromised solutions of equivalent quality, mostly known as the trade-off, non-dominated, non-inferior or Pareto-optimal solutions (Savic, 2002). Mathematically, the multi-objective optimization problem can be stated in its general form as:

$$\begin{aligned} &\min \mathrm{J}\,(\mathrm{x},\mathrm{p}) \\ &\mathrm{s.t.}\quad \mathrm{g}(\mathrm{x},\mathrm{p}) \leq 0 \\ &\qquad\;\; \mathrm{h}(\mathrm{x},\mathrm{p}) = 0 \end{aligned}$$

$$\mathrm{X}_{\mathrm{i.LB}} \leq \mathrm{X}_{\mathrm{i}} \leq \mathrm{X}_{\mathrm{i,UB}}\,(\mathrm{i}=1,\dots,\mathrm{n})$$
$$\mathrm{x} \in \mathrm{S}$$

Where

$$\begin{aligned} J &= [\,J_1(x)\dots J_z(x)\,]^T \\ x &= [x_1\,\dots\,x_i\,\dots\,x_n]^T \\ g &= [g_1(x)\,\dots\,g_{m_1}(x)]^T \\ h &= [h_1(x)\,\dots\,h_{m_2}(x)]^T \end{aligned}$$

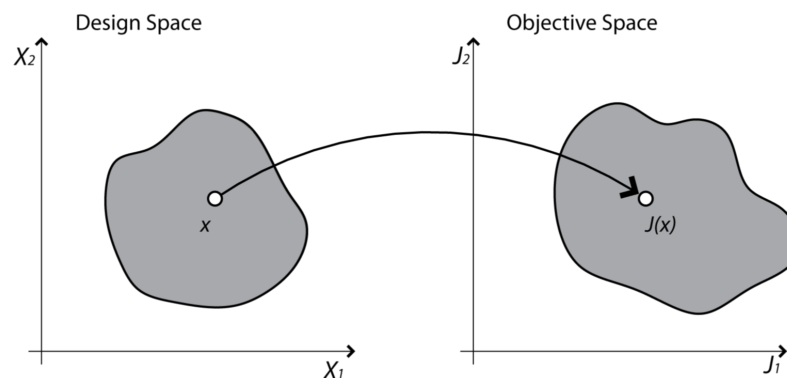
Here, J is a column vector of z objectives, whereby $J_i \in \mathbb{R}$. The individual objectives are dependent on the design vector x of n design variables. In order for a particular design x to be feasible, both a vector of inequality constraints g, and equality constraints h, have to be satisfied. The problem is to minimize (or maximize)

simultaneously all elements of the objective vector (de Weck, 2004).

Although the mathematical formulation of the optimization problem looks quite similar to single-objective optimization, they are considerably different. In multi-objective optimization, in addition to the design space in which each combination of design variables is available, a second space with the attainable objective function values exists, where a mapping process occurs for a design represented by the design vector *x* in the feasible design space *S* to the attainable objective space *J* (figure 5.36).

Figure 5.36:

Illustration of design space and objective space.



Multiobjective methodologies provide more realistic models of a problem because many objectives are considered and the emphasis on multiobjective thinking helps avoid potentially sub-optimal point designs (Cohon, 1978). Most design problems are characterized by a large and often infinite number of alternatives. A wider range of these alternatives is usually identified when multi-objective methodologies are implemented. (Cohon, 1978).

Although multiobjective optimization has been in use for some time now and its application in design problems has been continuously increasing, relatively few techniques have been developed compared to the large number of techniques available for single-objective optimization (Abraham et al., 2005).

5.5.3 Multiobjective Methods

Multiobjective optimization methods can be broadly classified into two categories: Decision making before search methods which are also known as Scalarization approaches, and Search before decision making methods which are also known as Pareto approaches. While different names are used for these categories, the fundamental differences are always the same (de Weck, 2004).

In the first category of methods the designer decides how to aggregate different objectives into a single objective function before the actual search is performed (Gries, 2004). This requires the formation of a single objective function that contains contributions from the sub-objectives in vector *J*. The formation of the aggregate objective function requires that the preferences or weights between objectives are assigned before the results of the optimization process are known. In this way, well-established single optimization methods can be applied (de Weck, 2004).

Table 5.2:

Different Scalarization and Pareto Methods (de Weck, 2004).

Scalaraization Methods (apriori preference expression)	Pareto Methods (a-posteriori preference expression)
• Weighted Sum Approach • Multiattribute Utility Analysis (MAUA) – Utility Theory. • Compromise Programming (Non-linear combinations). • Physical Programming, Goal Programming. • Lexicographic Approaches. • Acceptability Functions, Fuzzy Logic.	• Exploration and Pareto Filtering. • Multiobjective Genetic Algorithms (MOGA). • Weighted Sum Approach (with weight scanning). • Adaptive Weighted Sum method (AWS). • Normal Boundary Intersection (NBI). • Multiobjective Simulated Annealing (MOSA).

In the second category, the search for optimal solutions is performed with multiple objectives kept separate during the search. These Pareto methods typically use the concept of dominance to differentiate between inferior and non-inferior solutions. The result of the search is a set of Pareto-optimal solutions. Additional criteria or preferences can be applied after the search to find an optimal solution for a given problem. In this manner an unbiased search can be performed. In addition, a single search can serve several problem-specific decisions without the need to repeat the search (Gries, 2004).

Therefore, the selection of a single- or a multi-objective search algorithm influences not only the point of time when design objectives are defined, but also influences the whole exploration process (Gries, 2004). However the end goal of all these methods remains the same: to provide designers and decision makers with a set of alternatives to choose from (de Weck, 2004). Table 5.5.1 provides an overview of Multi-Objective Optimization Methods.

5.5.3.1 Decision Making before Search

In the decision making before search approach, multi-objectives are formulated into a scalar substitute problem that has a scalar objective and can be solved with the usual single objective optimization methods. This method is called scalarization and is based on the assumptions that the designer preferences are known and assigned before searching the design space for design solutions.

The scalar objective has the form *f*(*x*, *p*), where *p* is a vector of preference parameters that can be tuned to the designer's subjective preferences. The *z* objectives can be aggregated to express a utility, *U*, a dimensionless scalar quantity expressing the quality of a particular design.

$$\begin{aligned} &\max \{\mathrm{U}(\mathrm{J}_1, \mathrm{J}_2, \dots, \mathrm{J}_z)\} \\ &\mathrm{s.t.} \quad \mathrm{J}_\mathrm{i} = \mathrm{f}_\mathrm{i}(\mathrm{x}, \mathrm{p}) \quad 1 \le \mathrm{i} \le \mathrm{z} \\ &\qquad \mathrm{x} \in \mathrm{S}, \mathrm{U} \in \mathrm{R}^{+} \end{aligned}$$

Several scalarization methods have been developed (Table 5.5.1). The focus in the following will be on two of these methods namely the weighted-sum approach and the utility function approach.

5.5.3.1.1 Method of Weighted-Objectives

One of the most common and easiest to understand scalarization techniques is the weighted sum approach which is also known as the method of weighted-objectives. The scalar substitute objective is obtained by assigning subjective weights to each objective and summing up all objectives multiplied by their corresponding weight (Papalambros and Wilde, 2000).

The decision maker weights the different criteria according to their relative importance in determining the quality of a solution. This numerical treatment facilitates comparison among criteria that are not related (Kockler et al., 1990). Weighting should follow a logical breakdown.

This approach is characterized by one composite or utility function *U* declared by aggregating multiple objective functions with individual weighting factors *λj*.

$$\begin{aligned} &\max \qquad \mathrm{U}\left(\mathrm{J}(\mathrm{x}, \mathrm{p})\right) \\ &\text{where} \quad \mathrm{U} = \sum_{\mathrm{j}=1}^{\mathrm{z}} \lambda_\mathrm{j} \frac{J_j}{sf_j} \text{ with } \lambda = [\, \lambda_1 \;\; \lambda_2 \;\; \dots \;\; \lambda_z \,]^{\mathrm{T}} \end{aligned}$$

$$\text{and} \qquad \lambda \in \mathrm{R}^{z} \quad | \quad \lambda_i > 0, \sum_{i=1}^{z} \lambda_i = 1$$

$$\text{and} \qquad x \in S$$

Formulated in this manner the objective *U* always forms a strictly convex combination of objectives. The individual objectives are typically normalized, and since the optima of the problem does not change if all weights are multiplied by a constant value, weights are chosen such that they add to unity and are themselves positive scalars (de Weck, 2004).

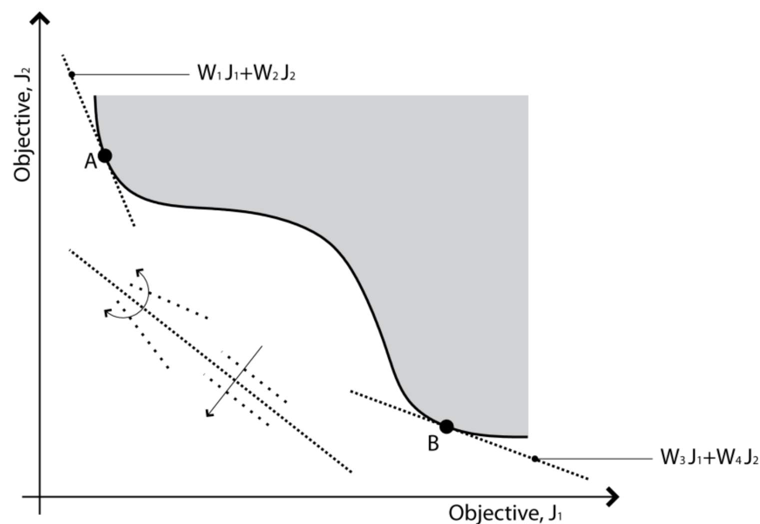
It is apparent that the preference of an objective can be changed by modifying the corresponding weighting factor which leads to another solution point. In the case of two equally scaled objectives:

$$U = \lambda J_1 + (1 - \lambda) J_2$$

The ratio of the weights defines the constant slope of the line. Varying λ gradually in small incrementing steps exposes a set of optimal solutions as the weight is gradually shifted from one objective to another. This sequential variation of some weighting factors can be used to find as much trade-off solutions as possible.

Figure 5.37:

Sequential variation of weighting factors can be used to find trade-off solutions.



This approach can be utilized to find the Pareto-front by obtaining different points on the curve with different combinations of weighting factors (figure 5.37). Although this approach can work for convex Pareto-fronts, it does not work for non-convex cases since not all points on the Pareto-front can be determined. It is apparent from figure 5.38 that many points in the non-convex case will never

be reached with any combination of the weights and the resulting optima are unevenly distributed.

5.5.3.1.2 Utility

Another scalarization approach is the utility functions approach which is based on the general formulations of *utility theory.* Utility functions may be developed using engineering judgment or a more quantitative approach. The range of the utility function covers a range of acceptable alternatives. Most scalarization approaches can be represented via the utility function approach (de Weck, 2004).

A mathematical construction of a utility function allows non-linear combinations of objectives via intermediate utility functions, which are then combined into an overall utility function that will serve as a single objective. The method assigns costs to each objective, converting everything to minimum cost (Papalambros and Wilde, 2000). The method normalizes the utility functions. This provides for a mediating capability by translating diverse criteria into a common scale (Kockler et al., 1990).

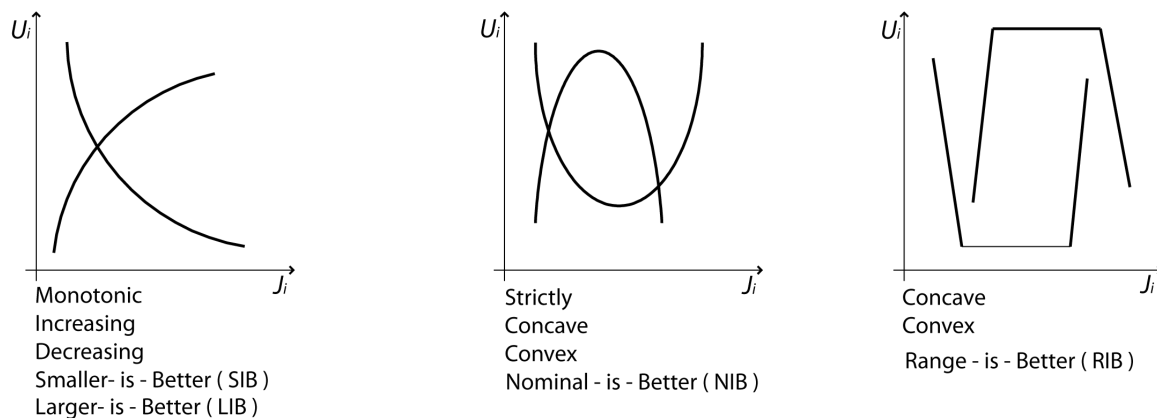


Figure 5.38:

Different utility functions classifications (Cook 1997, Messac 2000).

Utility functions have been classified by various researchers into their most prevalent shapes (Cook 1997, Messac 2000). For example a larger-is-better or smaller-is-better relationship is represented by a monotonically increasing or decreasing relationship between the objective *Ji* and its corresponding utility *Ui,* whereas a nominal-is-better or in range-is-better type of utility can be represented by a convex or concave functions (figure 5.38).

Although utility functions are effective and commonly used they are hard to implement and require extensive interviews to determine

appropriate utility functions and weights. However, once the utility functions have been constructed, optimization can be performed in search of the design with maximal utility (de Weck, 2004).

Scalarization Methods Discussion

In the previously discussed scalarization methods, multiple objectives are aggregated into one objective using some knowledge of the design problem being solved. Using scalarization methods and the optimization of a single-objective may only provide a single Pareto-optimal solution point for a convex Pareto-front. However, designers usually need different alternatives and need to carry out trade-offs between different objectives. Therefore, in order to increase the number of points on the Pareto-front, the same problem can be solved several times with variable weight settings. However, this process may not work effectively for a non-convex Pareto- front.

Furthermore scalarization methods may include some subjective information and therefore may be misleading in regards to the character of the optimum design solution (Papalambros and Wilde, 2000). In addition, solutions obtained using scalarization methods mainly depend on the method settings such as the underlying weight-vector for the weighted sum approach or the manner in which the utility interviews were conducted.

Another weakness of these methods is that they require knowledge of the optima prior to starting the optimization, but design preferences are rarely known precisely a priori. Shifts in preference values can occur once the set of feasible designs becomes known. Therefore trade-offs become more evident with time.

Nonetheless, scalarization methods are useful in gaining fast single solutions especially if a single-objective optimization method is the only available method (Keskin, 2007).

5.5.3.2 Search Before decision making

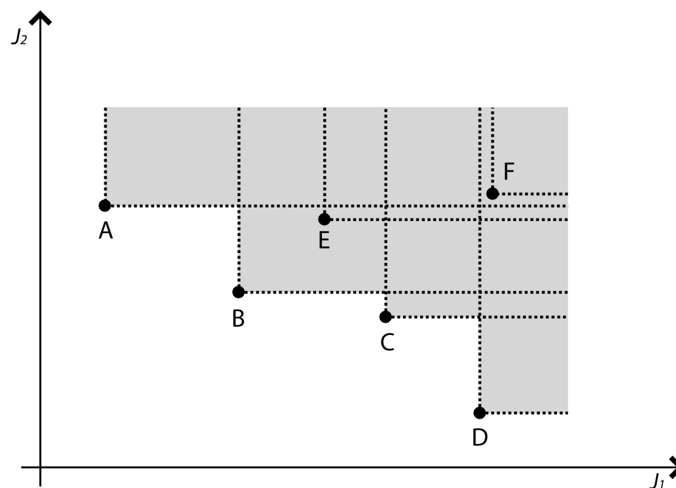
One of the first scientists to introduce the concept of trade-offs between objectives was F.Y Edgeworth in 1881. Edgeworth was a Professor of economics and defined an optimum for multicriteria economic decision-making (Edgeworth 1881). He did so for the multi-utility problem within the context of two hypothetical consumer criteria, P and π: *"It is required to find a point (x,y,) such that in whatever direction we take an infinitely small step, P and π do not increase together but that, while one increases, the other decreases."*

Vilfredo Pareto (1848-1923), a contemporary of Edgeworth who was working in Florence as a civil engineer, was one of the first to analyze economic problems with mathematical tools. His concept, named the Pareto Optimum, found broad acceptance (Pareto 1906): *"The optimum allocation of the resources of a society is not attained so long as it is possible to make at least one individual better off in his own estimation while keeping others as well off as before in their own estimation."* Since then, multi-objective optimization has penetrated design and engineering and has developed at a rapidly increasing pace (de Weck, 2004).

A solution may be better, worse or indifferent to other solutions, neither dominating nor dominated with respect to the objective values (Abraham et al., 2005). In a multi-objective optimization problem there exists a set of solutions which are superior to the rest of the solutions in the search space when all objectives are considered but inferior to other solutions in the space in at least one objective. These are optimal solutions that are not dominated by any other solution in the search space (figure 5.39). Such optimal solutions are called Pareto optimal, and the entire set of such optimal trade-offs solutions is called the Pareto optimal set, where the rest of the solutions are called dominated solutions (Abraham et al., 2005).

Figure 5.39:

Points A,B,C and D are optimal solutions that are not dominated by any other solution in the search space.



All elements in the Pareto optimal set define reasonable solutions and are subject to further decision factors in order to choose a design for a given problem (Gries, 2004). As evident, in a real world situation a decision-making (trade-off) process is required to obtain the optimal solution (Abraham et al., 2005).

Although there are several methods to approach a multiobjective optimization problem, most work is concentrated on the approximation of the Pareto set (Abraham et al., 2005).

Pareto methods attempt to find a set of efficient solutions, x^{*j} , such that the objective vectors corresponding to those solutions are non-dominated in the objective space.

To explain the Pareto criterion for dominance we will assume, without loss of generality, two feasible objective vectors J^1 and J^2. For all objectives, respectively, J^1 dominants J^2 if and only if:

$$J_i^1 \geq J_i^2 \quad \forall\, i$$

$$\text{And} \quad J_i^1 \geq J_i^2 \quad \text{for at least one } i$$

This means a dominant solution is at least better in one objective while being at least the same in all other objectives. For strong (strict) dominance requires J^1 to be better in all objectives than J^2 .

Based on the notion of dominance, the simplest approach address the multi object decision is a combination of design space exploration and dominance (Pareto) filtering.

The advantage of multi-objective optimization compared to single objective optimization is to provide different solutions to the design problem that the designer can choose from. To pick one solution over another might require problem knowledge and additional decision criteria which are not necessarily formulated in the design task. Therefore, it may be useful to have a wide range of non-dominated solutions from which one or more solutions can be chosen.

Two goals can be pursued simultaneously in multi-objective optimization (Deb 2001). The first goal is to find a diverse set of solutions. However, this set won't be comprehensive due to the *n*-dimensionality of the design vector *x*.

The Second goal is to find a set of solutions as close as possible to the Pareto-optimal front. Given that the points only satisfy non-dominance, the solutions obtained are only approximations of the Pareto Front.

An optimum to the problem is found if they satisfy the multi-objective version of the Karush-Kuhn-Tucker (KKT) optimality conditions (de Weck, 2004):

If x^* is non-inferior (=Pareto optimal) it satisfies the following KKT conditions:

a) x^* is feasible, i.e. $x^* \in S$ and $S = \emptyset$
b) All objective functions J_i and constraints g_j are differentiable
c) At x^* the constraints are satisfied $g_i(x^*) \leq 0 \; \forall \; j = 1,2, \dots, m$ and $\lambda_j g_j \; (x^*) = 0$ whereby $\lambda_j \geq 0 \; \forall \;\; j = 1, \dots , m$
d) There exist $\mu_i \geq 0 \; \forall \; i = 1, \dots \; , n$ with strict inequality holding for at least one i such that the condition $\sum_{i=1}^{n} \mu_i \, \nabla J_i(x^*) + \sum_{j=1}^{m} \lambda_j \, \nabla_{g_j}(x^*) = 0$ is true.

The condition described in (d) expresses the fact that the gradients of the objectives and gradients of the constraints are in equilibrium with each other at a Pareto-optimal point. Note, that among multipliers, the preferences μi are the corollary to the weights (λi), while the λ *j*'s are the Lagrange multipliers.

5.5.3.2.1 MOGA

Several stochastic optimization techniques such as simulated annealing; tabu search, ant colony optimization etc. could be used to generate the Pareto set. However, due to the manner that these algorithms work, the solutions generated tend to be stuck at good approximations and do not guarantee the identification of optimal trade-offs (Abraham et al., 2005)

In recent years there has been a rising interest in evolutionary multiobjective optimization algorithms which combine both evolutionary computation and multicriteria decision making. One of the strongest appeals of such algorithms is that they require very little knowledge about the design problem being solved, and are easy to implement, robust and could be implemented in a parallel environment (Abraham et al., 2005)

Several evolutionary algorithms have been proposed and successfully applied to various design problems. The *Multi-Objective Genetic Algorithm* (MOGA) proposed by Fonseca and Fleming (1993) has particularly gained increased acceptance among the Pareto approaches in recent years.

MOGA's evolve populations of designs gradually so that they approximate a Pareto frontier as closely as possible (de Weck, 2004). The approach consists of a scheme in which the rank of a certain individual corresponds to the number of individuals in the current population by which it is dominated. All non-dominated individuals are assigned rank 1, while dominated ones are penalized according to

the population density of the corresponding region of the trade-off surface (Coello, 2001). Fitness assignment is performed in the following way (Fonseca and Fleming, 1993):

1. Sort population according to rank.

2. Assign fitness to individuals by interpolating from the best (rank 1) to the worst (rank $n < M$) in the way proposed by Goldberg (1989) (the so-called Pareto ranking assignment process), according to some function, usually linear, but not necessarily.

3. Average the fitnesses of individuals with the same rank, so that all of them will be sampled at the same rate. This procedure keeps the global population fitness constant while maintaining appropriate selective pressure, as defined by the function used.

Since the use of a blocked fitness assignment scheme as the one indicated before is likely to produce a large selection pressure that might produce premature convergence (Goldberg, 1989), the use of a niche-formation method proposed to distribute the population over the Pareto-optimal region (Deb and Goldberg, 1989). Sharing is performed on the objective function values. MOGA also uses mating restrictions.

While MOGA is an effective method and doesn't require apriori assignments of weights, there are a few challenges (de Weck, 2004). The main challenges are the large computational expense as well as a tendency for niching (clumping of solutions in objective space) which results in underrepresented regions of the Pareto front (de Weck, 2004). There is a need to minimize the distance of the generated solutions to the Pareto set and to maximize the diversity of the developed Pareto set. A good Pareto set may be obtained by appropriate guiding of the search process through careful design of reproduction operators and fitness assignment strategies. To obtain diversification special care has to be taken in the selection process. Special care is also to be taken care to prevent non-dominated solutions from being lost (Abraham et al., 2005). All of these issues (and others) are subjects of ongoing research in the multiobjective optimization community.

5.6 Optimization Models

5.6.1 What is an Optimization Model?

Optimization was first coined with the development of the gradient steepest descent algorithm by Guass. It served as the first building block of the science of optimization. Later, in the 1940s, George Dantzig invented the term linear programming which gave way to the development of the remaining well-known optimization schemes (Elster, 1993). Throughout the 1970s and 1980s, the field of Artificial Intelligence (AI) introduced the heuristic approach to solving optimization problems. Today, optimization plays an important role in most of the major fields, which include engineering and design, operations research and economics.

Conventional design procedures are for finding a suitable design which satisfies the functional objective(s) and requirements of the problem. In general, there will be more than one acceptable design or design alternative. The purpose of optimization is to evaluate and choose the fittest of the available acceptable designs based on the functional objective(s) and the design requirements and restrictions.

Optimization can be explicated as improving or fine-tuning a design or system in terms of one or more performance criteria (Papalambros, 2000). It formalizes what humans have always done intelligently. Optimization can be used in refining any design or system that includes some form of an analysis component, and is therefore subjected to the same limitations of the design. Generally, an optimization problem consists the following (Papalambros and Wilde, 2000):

- A set of variables that describe the design alternatives.
- An objective function(s), expressed by the design variables, to minimize or maximize.
- A set of constraints, expressed in terms of the design variables, to be satisfied by any suitable design.
- A set of values for the design variables, which satisfies all the constraints.

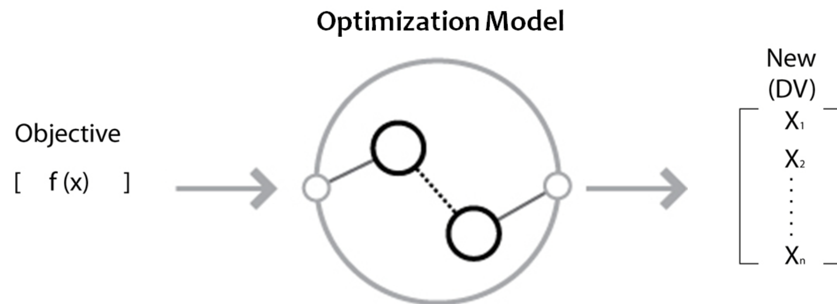
Certain design features are determined in the synthesis model, and the behavior corresponding to each design is determined in the analysis model. The evaluation model attempts to handle the multi-objective criteria of the design problem. Optimizing models are then used to determine optimal designs.

The input to an optimizing model is an objective function. This could

be the output of an evaluation model. The output of an optimization model is a new design vector that in turn is the input to the synthesis model (figure 5.40).

Figure 5.40:

Expected input and output of the optimization model.



The rising demand for industry to lower production costs has encouraged professionals to seek precise and accurate means for decision-making, leading them to utilize new optimization schemes. Optimization methods today have reached a high degree of sophistication, contributing to their use in a wide range of industries. With the rapid advancement of computer technology, the size and the complexity of the problems being solved using optimization techniques are also increasing.

In this section, optimization will be discussed in general. I will start by explaining how optimization problems are mathematically formulated and classified. Next, the main optimization algorithms will be discussed.

5.6.2 Mathematical Formulation

Mathematically, optimization is the minimization or maximization of a function subject to constraints on its variables (Nocedal and Wright, 2000). The objective function is sometimes called a "cost" function, since minimum cost is often taken to characterize the "best" design. In general, the criterion (objective function) for selection of the optimal design is a function of the design variables of the model.

The following notations are used to represent a model (Papalambros and Wilde, 2000):

- x *is* the vector of *variables*, also called *unknowns* or *parameters.*
- f is the *objective function*, a (scalar) function of x that is

maximized or minimized.

- c_j are *constraint* functions, which are scalar functions of *x* that define certain equations and inequalities that the unknown vector *x* must satisfy.

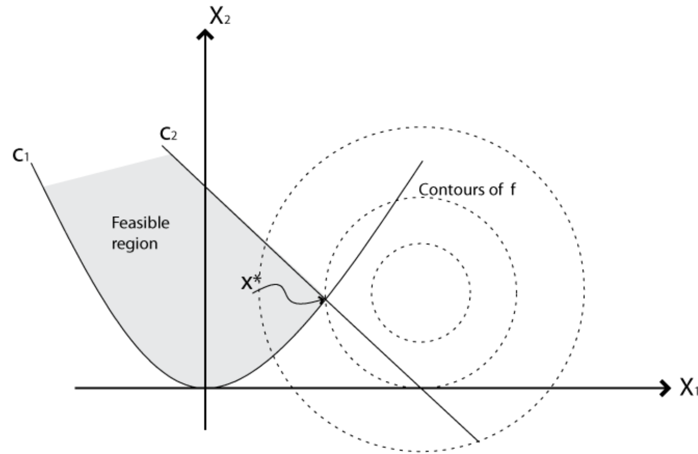


Figure 5.41:

An optimization problem has an objective function and can have several constraints to insure feasibility.

Using this notation, the optimization problem can be written as follows:

$$\min_{x \in R^n} f(x) \text{ subject to}$$

$$c_j(x) \geq 0, \quad i \in I$$
$$c_j(x) = 0, \quad i \in E$$

Here *E* and *I* are sets of indices for equality and inequality constraints, respectively.

The variables are expected to be interrelated by physical laws, like the conservation of mass or energy, Kirchhoff's voltage and current laws, or other system equalities that must be satisfied (Antoniou et al., 2007). Similarly a collection of constraints may be imposed on the variables to ensure physical reliability, compatibility, or even to simplify the modeling of the problem (Nocedal and Wright, 2000)(figure 5.41).

5.6.3 Classification of Optimization Problems

The classification of an optimization problem depends on more than one factor. It can be the objective function, constraints, design variables etc. This will be discussed further in the following.

5.6.3.1 Based on Constraints

Depending on the ranges allowable for the design variables, optimization problems can be classified as unconstrained and constrained optimization problems.

Unconstrained optimization

In some problems there might be no need to have constraints on the variables, or it might be safe to ignore them as they do not have an effect on the solution and do not interfere with algorithms. Unconstrained problems can also arise from reformulations of constrained optimization problems, in which the constraints are replaced by penalization terms added to the objective function, that have the effect of restricting constraint violations (Nocedal and Wright, 2000).

Mathematically an unconstrained problem is represented as:

$$\text{Find } \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ which minimizes } f(\mathbf{X})$$

Constrained optimization

These kinds of problems come up from models in which constraints play an essential role i.e. they are necessary, for example in imposing budgetary constraints in an economic problem or shape constraints in a design problem (Nocedal and Wright, 2000).

Mathematically a constrained problem can be stated as follows:

$$\text{Find } \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \text{ which minimizes } f(\mathbf{X})$$

Subject to the constraints

$$g_j(X) \geq 0, \qquad j = 1,2, \ldots \ldots, m$$
$$l_j(X) = 0, \qquad j = 1,2, \ldots \ldots, p$$

Where X is an n-dimensional vector called the *design vector, f*(X) is the *objective function, and the* constraints are, g_j (X) for inequality, and l_j (X) for *equality* (Rao, 1996).

5.6.3.2 Based on Design Variables

Depending on the values allowable for the design variables, optimization problems can be classified as integer (Discrete) and real-valued (Continuous) programming problems.

Models with continuous variables are in general easier to solve with techniques based on differential calculus. However, models with discrete variables are combinatorial in nature and an optimal solution is difficult to identify without sometimes complete enumeration of all possible combinations which is not always practical (Papalambros, 2000).

Discrete Variables

If some or all of the design variables $x_1, x_2, \ldots, x_n$ are restricted to take on only integer or discrete values, then the problem is called an *integer programming problem.* The behavior of the objective function and constraints may change as we move from one possible point to another, even if the two points are "close" by some measure (Nocedal and Wright, 2000).

Continuous Variables

Similarly, if all the design variables can take any real value, the optimization problem is called a *real-valued or continuous programming problem.* These problems are in general easier to solve because of the smoothness of the function, which makes it feasible to use the objective function and constraint information at a particular point *x*, to realize the function's behavior at all points close to *x*. (Nocedal and Wright, 2000)

5.6.3.3 Based on nature of objective function

If the modeling relationships and the objective function are linear, the optimization problem is linear. This type of problem is usually referred to as a linear programming problem. If either the modeling relationships and/or the objective function are nonlinear the optimization problem is nonlinear, or a nonlinear programming problem.

Frequently very large-scale nonlinear or linear programming problems are decomposed into a set of interconnected (and in some sense) simpler problems (Papalambros, 2000).

Linear programming

When the objective function and all its constraints, are linear in terms of *x*, the problem is called a *linear programming* problem. These types of problems are perhaps the most widely formulated and solved of all optimization problems, mainly in management, financial, and economic applications. (Nocedal and Wright, 2000)

The mathematical representation of a *linear programming* problem in standard form:

$$\text{Find } \mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

This minimizes

$$f(\boldsymbol{X}) = \sum_{i=1}^{n} c_i x_i$$

Subject to the constraints

$$\sum_{i=1}^{n} a_{ij} x_i = b_i, \qquad j = 1,2,\ldots\ldots,m$$
$$x_i \geq 0, \qquad i = 1,2,\ldots\ldots,n$$

Where c_i, b_j, and a_{ij} are known constants, and x_i are the decision variables. (Rao, 1996)

Nonlinear programming

Problems, in which at least some of the constraints or the objectives are nonlinear functions, are called *Nonlinear programming* problems. They tend to occur naturally in the physical sciences and engineering, and are becoming more widely used in management and economic sciences as well (Nocedal and Wright, 2000).

A nonlinear programming problem with a quadratic objective function and linear constraints is considered a Quadratic Programming Problem. Mathematically it is formulated as follows:

$$F(X) = c + \sum_{i=1}^{n} q_i x_i + \sum_{i=1}^{n} \sum_{j=1}^{n} Q_{ij} x_i x_j$$

Subject to

$$\sum_{i=1}^{n} a_{ij} x_i = b_i, \qquad j = 1,2,\ldots\ldots,m$$
$$x_i \geq 0, \qquad i = 1,2,\ldots\ldots,n$$

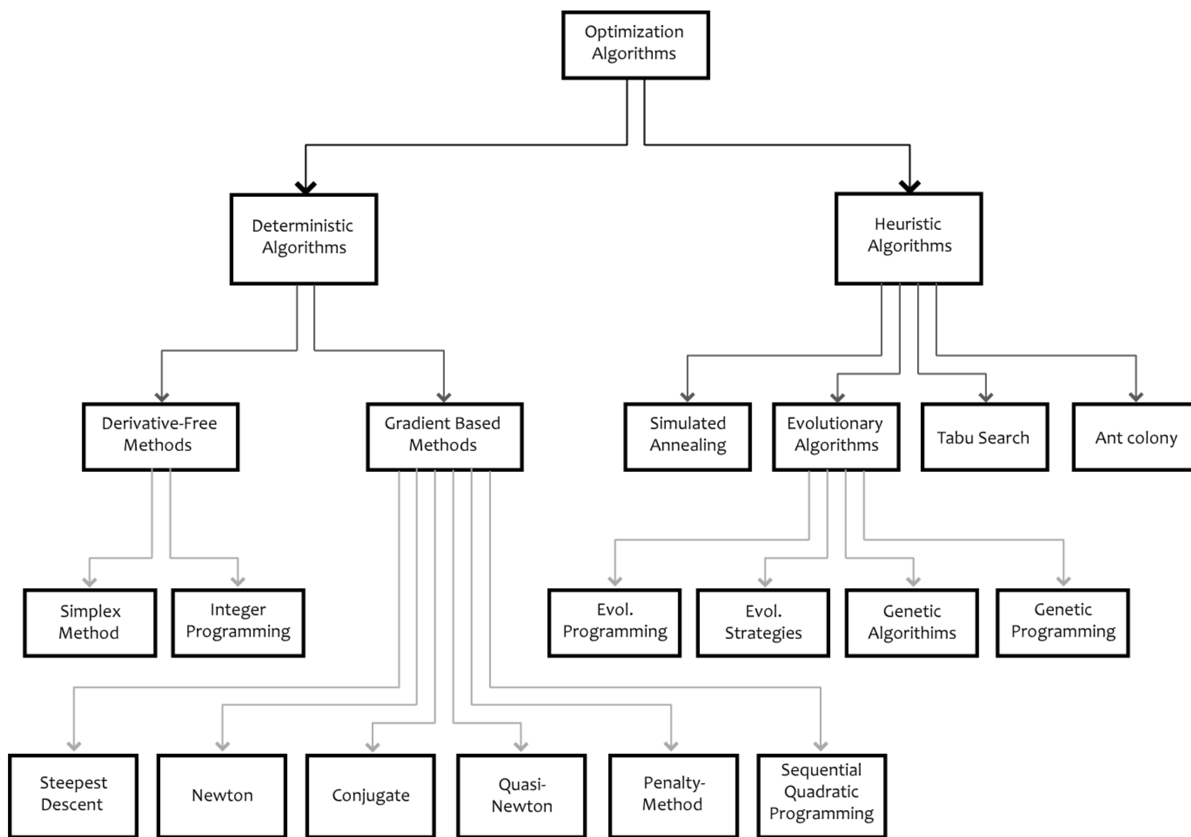
Where, c, q_i, Q_{ij}, a_{ij}, and b_j are constants (Rao, 1996).

5.6.4 Classification of Optimization Algorithms

Figure 5.42:

A simple taxonomy of optimization algorithms discussed in the thesis.

Optimization Algorithms can be classified into either Deterministic or Stochastic (Heuristic) methods. Deterministic methods can be classified into Derivative-Free methods and Gradient Based methods. Stochastic (Heuristic) methods include several algorithms such as Evolutionary Algorithms, Simulated Annealing and Tabu Search (figure 5.42). In the following sections emphasis will be given to some better known optimization algorithms.



5.6.4.1 Deterministic Algorithms

5.6.4.1. 1 Derivative-Free Methods

Simplex Method

Linear programming and the Simplex method have been the most widely used amongst optimization tools since the 1950s (Luenberger, 2003). Linear programs have linear objective functions and constraints that include both equalities and inequalities. The standard form of linear programs is:

$$\min\ c^T x,$$
$$subject\ to\ Ax = b, x \geq 0,$$

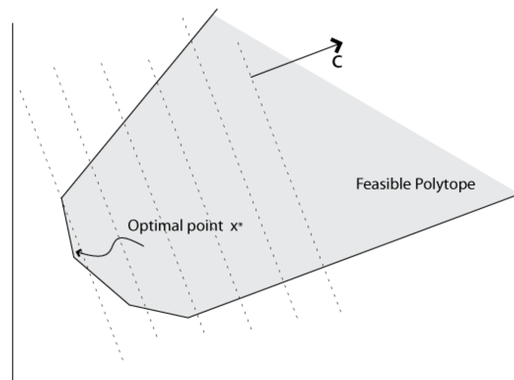
Where *c* and *x* are vectors in R^n, *b* is a vector in R^m, and *A* is an *m* × *n* matrix.

In geometric terms, the feasible set defined by the linear constraints is a polytope, an n-dimensional convex, defined by the intersections of a number of half-spaces in *n*-dimensional Euclidean space (Luenberger, 2003).

The *vertices* of this polytope are the points that do not lie on a straight line between two other points in the set (Nocedal and Wright, 2000). Algebraically, the vertices are exactly the basic feasible points defined above (figure 5.43). The contours of the objective function are planar and the set of optimal points can be a single vertex, an edge or face, or an entire feasible set (Luenberger, 2003).

Figure 5.43:

Linear program in two dimensions with solution at $x^*=c^Tx$



Simplex Method Algorithm

The idea of the simplex method is to proceed from one basic feasible solution of a problem in standard form to another, in a manner that continually decreases or increases the value of the objective function until an optimum is reached (Luenberger, 2003). The changing of a non-basic variable value and adjusting the values of the basic variables, while maintaining feasibility, corresponds to moving along an edge of the convex set (Dantzig, 1998).

Therefore, interior points can be ignored while focusing only on the edges, because the simplex algorithm attempts to find a single optimal point. Hence, the simplex algorithm begins at a vertex and moves along the edges of the polytope until it reaches the vertex of the optimum solution. The Simplex Method provides an efficient method for moving among basic solutions to an optimal solution (Luenberger, 2003).

To apply the simplex algorithm, the linear programming problem has to be transformed into the augmented form. The optimization problem is formulated in matrix form as follows:

Maximize Z in:

$$\begin{bmatrix} 1 & -c^T & 0 \\ 0 & A & I \end{bmatrix} \begin{bmatrix} Z \\ X \\ X_S \end{bmatrix} = \begin{bmatrix} 0 \\ b \end{bmatrix}$$

$$X,\ X_S \geq 0$$

where Z are the variables to be maximized, x are the variables from the standard form, x_s are slack variables from the augmented form, c contains the optimization coefficients, A and b describe the constraints.

This form helps defining the initial basic feasible solution. All variables from the standard form are nonbasic variables that have a zero value, whereas the new variables introduced in the augmented form are basic variables that have a nonzero value.

The algorithm starts at some vertex of the polytope and at every iteration selects an adjacent vertex that does not decrease the value of the objective function. If no such vertex exists then a solution to the problem is found. However often an adjacent vertex is not unique and a pivot rule must be applied to establish the next vertex to select.

Revised Simplex method Algorithm

Rather than spend time updating tableaus and dictionaries at the end of each iteration, the Revised Simplex Method does most of its calculation at the beginning of each iteration which results in less calculation at the end.

However in computer implementation, the physical limitations of the computer can become an issue since round-off errors are a common problem in matrix manipulations particularly since matrices generated from the Revised Simplex Method are not usually well-conditioned. Therefore, the task of implementing the Revised Simplex Method is more than just coding and programming the algorithm but is also an exercise in numerical stability.

Duality Theory

For every linear programming problem, also known as a primal problem, there is a corresponding dual linear programming problem that provides an upper bound on the optimal value of the primal problem (Luenberger, 2003).

If the primal problem is formulated in matrix form as follows

Maximize $c^T x$
subject to $Ax \leq b,\ \ x \geq 0$

Then the corresponding dual problem is formulated in matrix form as follows:

Minimize $b^T y$
subject to $A^T y \geq c,\ \ y \geq 0$

where *y* is used instead of *x* as variable vector.

Both problems are constructed from the same underlying cost and constraint coefficients but in a manner that if one of these problems is minimized then the other is maximized, and if the optimal values of the corresponding objective functions exist, then they are equal (Luenberger, 2003).

The variables of the dual problem can be interpreted as prices associated with the constraints of the original (primal) problem. Through this association it is possible to give an economically meaningful characterization to the dual. Furthermore, the variables of the dual problem are also related to the calculation of the relative cost coefficients in the simplex method. The consideration of the linear programming problem from both the primal and dual

viewpoints usually provides insight and significant computational advantage (Luenberger, 2003).

There are two important duality theorems that should also be discussed: the weak duality theorem and the strong duality theorem. The weak duality theorem states that the value of the objective function of the dual is always greater than or equal to the value of the objective function of the primal. On the other hand, the strong duality theorem states that if the primal has an optimal solution x*, then the dual also has an optimal solution y*, such that $c^T x^* = b^T y^*$.

Integer Programming

When formulating a Linear Programming problem, if certain variables need to be integer values then we are faced by a more difficult type of problem called integer programs (IP's).

Integer programs occur frequently because many decisions are essentially discrete. A pure integer programming problem is one where all variables in the integer programming problem are integers. A mixed integer programming problem is one where only some variables are restricted to being integers. A binary integer programming problem is one where the integer variables are restricted to be 0 or 1.

There are two well-known algorithms for solving integer programming problems, namely the Branch and Bound algorithm and the cutting plane algorithm. The Branch and Bound algorithm is based on dividing the problem into a number of smaller problems. The cutting plane algorithm, on the other hand, is based on adding constraints to force integrality.

Both methods involve solving a series of linear programs by first solving a relaxed version of the problem, and then adding constraints until an integer solution is found. Given the integer program:

$$\begin{gathered}\text{Minimize (or maximize) } ci \\ \text{subject to } Ai = b \\ i \geq 0 \text{ and integr}\end{gathered}$$

It’s associated linear relaxation:

$$\begin{gathered}\text{Minimize (or maximize) } ci \\ \text{subject to } Ai = b \\ i \geq 0\end{gathered}$$

The linear relaxation problem is formed by dropping the integrality constrains and therefore is less constrained than integer programming.

Solving a linear relaxation problem provides some information about the problem and sets a bound on the optimal value. However, it must be noted that rounding the solution of linear relaxation is not expected to produce the optimal solution of an integer program and that other techniques have to be implemented.

Branch and Bound

The Branch and Bound method was first introduced by Land and Doig in the early 1960's. The algorithm solves an integer programming problem by enumerating feasible solutions such that the optimal integer solution is found. However, unlike a complete enumeration, this method does not consider each possible enumeration because that is likely to be too large. This is an important factor that enables the tree search to work and find solutions that would be very hard if complete enumeration was used.

A branch-and-bound method requires two procedures. The first is a branching procedure that given a set of candidates returns two or more smaller sets. This procedure is a recursive procedure that defines a tree structure whose nodes are the subsets of the original set. The second procedure is a bounding procedure that computes upper and lower bounds of the given a subset.

The way the algorithm works is by first solving the relaxed version of the problem. If the solution is not an integer, then the branching procedure is implemented splitting the problem into two sub-problems. If while solving any of the sub-problems an integer solution is found, the nodes of the enumeration tree that are descendents of the current node are pruned since no better solution will be found by branching into even more sub-problems (Wolsey and Nemhauser, 1999). The method is repeated until there are no active sub-problems.

The Cutting Plane Algorithm

An alternative to the branch and bound method is the cutting planes method which can also be used to solve integer programs.

The cutting plane algorithm works by first finding the solution to the relaxed version of the problem (Papadimitriou and Steiglitz, 1998).

The fundamental idea behind cutting planes is to add constraints to a linear program until the optimal basic feasible solution takes on integer values.

Care however should be taken while adding the constraints in a manner that does not change the actual problem. A cut, which is a special constraint that is relative to a current fractional solution, is added such that every feasible integer solution has to be feasible for the cut, but that the current fractional solution is not feasible for the cut. If the solution found has non-integer elements, then new cuts are imposed that eliminate the previously found solution, but do not eliminate any feasible integer solutions, until a solution with all integer elements is found (Papadimitriou and Steiglitz, 1998).

One of the known methods used to generate cutting planes is called the Gomory cuts which can generate cuts from any linear programming tableau. This method's strength is in its ability to solve any integer programming problem. However its weakness is its slowness. Another approach to generating cutting planes depends on understanding the structure of the optimization problem to generate efficient cuts. Although this could provide powerful methods, it is problem specific.

In general, the cutting plane algorithms suffer from two main disadvantages. First, difficulties caused by round-off errors. And second, the large number of constraints generated. These two disadvantages are enough to render the cutting plane algorithm unfeasible. Nevertheless this method, combined with the branch and bound method could be a robust method in special types of problems.

In summary, the Integer programming is an NP-hard problem and is likely to remain so for the foreseeable future. In this section two algorithms for solving integer programming problems were discussed briefly namely the branch and bound algorithm and the cutting planes algorithm. Both algorithms are based on repetitively solving relaxed versions of the original problem and modifying it until an integer solution is found.

For both algorithms there is no guarantee on the time needed to reach a solution. In a worst case scenario, it is possible that we could build the entire enumeration tree using branch and bound. Based on the structure of the problem, the cutting plane algorithm may require many iterations and numerous cuts before arriving at an integer solution.

5.6.4.1. 2 Gradient Methods

Gradient methods depend on gradient information. Originally they were developed for unconstrained optimization problems. Variations of these methods were also developed to handle constrained optimization problems. The following sections will discuss both approaches.

Unconstrained Gradient Methods

Gradient methods are generally grouped into two categories, first-order and second-order methods. First-order methods are based on the linear approximation of the Taylor series, and therefore only require gradient information. Second-order methods, on the other hand, are based on the quadratic approximation of the Taylor series and require both the gradient and the hessian (Antoniou et al., 2007).

There are several gradient methods that range in their sophistication. In this section, some of the more basic methods will be presented namely: the steepest-descent method and the Newton method.

Steepest Descent

The steepest-descent method is a first-order method since it is based on the linear approximation of the Taylor series. The steepest-descent method is also considered the simplest of the gradient methods.

To find a local minimum of a function f using steepest descent, we need to choose the direction *d* where f decreases most rapidly. Assuming that a function $f(x_i)$ is continuous in the neighborhood of point x_i. We take steps proportional to the opposite *(negative)* of the gradient $\nabla f\ (X_i)$ of the function at the current point x_i. Generally *d* does not point in the direction of x* (local optimum) and therefore an iterative procedure must be used for the solution.

The search starts at an arbitrary point X_0 for a local minimum of f, and considers the sequence $X_0, X_1, X_2, \ . \ . \ .$ formally, the iterative procedure is

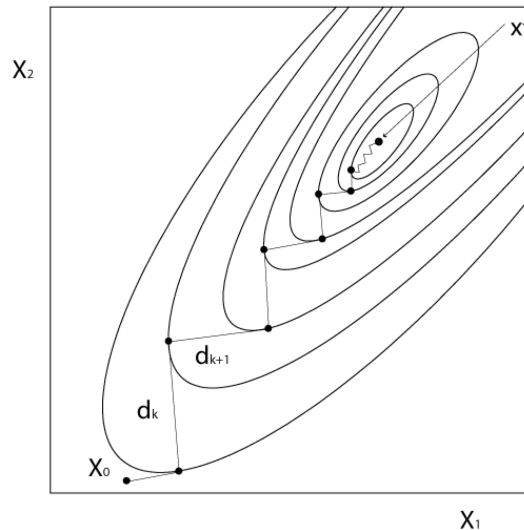
$$X_k + 1 = X_k - \lambda_k \nabla f(X_k)$$

$$k = 1, 2, \ldots n$$

With any luck the sequence (X_k) converges to the desired local minimum as we move down the gradient until we are close enough to the solution.

Figure 5.44:

In the steepest descent the trajectory to the solution follows a zigzag pattern



Note that the value of the *step size* λ_k is allowed to change at every iteration. Obviously, we want to move to the point where the function f takes on a minimum value, which is where the directional derivative is zero. The choice of λ_k is made such that the successive directions are orthogonal. The next step is taken in the direction of the negative gradient at this new point. This implies a minimization problem along a line, where the line equation is given by $X_k + 1 = X_k - \lambda_k \nabla f(X_k)$ for different λ_k values. This is solved by a *line search* for a minimum point along a line.

The iteration is repeated until the local minimum has been determined within a chosen accuracy ε. The trajectory to the solution follows a zigzag pattern (figure 5.44).

The Steepest Descent method is a simple, easy to apply, and fast method. It is also fairly stable. However the method has a few disadvantages. It generally has slow convergence and it is also highly dependent on a good starting point.

Newton Method

Unlike the steepest-descent method which is a first-order method based on the linear approximation of the Taylor series, the Newton method is a second-order method developed by using the quadratic approximation of the Taylor series of a given function f(x). The

information of the second derivative is used to locate the minimum of the function *f*(x). This is repeated in each iteration till the minimum is reached.

Any quadratic function has a Hessian which is constant for any point x. The quadratic function for x in an appropriate neighborhood of the current point X_k is given by a truncated Taylor series:

$$f(x) \approx f(X_k) + (X - X_k)^T . g_k + \frac{1}{2}(X - X_k)^T . H_k . (X - X_k)$$

Where both the gradient g_k and the Hessian matrix H_k are evaluated at X_k . If we take the derivative of this we get:

$$\nabla f(X) = g_k + \frac{1}{2} H_k . (X - X_k) + \frac{1}{2} H_K^T . (X - X_k)$$

If the function X_k is twice continuously differentiable at every point then the Hessian matrix is symmetric. Therefore we get

$$\nabla f(X) = g_k + H_k . (X - X_k)$$

If x* is the minimum of *f*(x) then the gradient is zero:

$$(X^* - X_k) + g_k = 0$$

x* is considered the next current point, resulting in the iterative formula:

$$\mathrm{X_{k+1} = X_k - H_k^{-1} . g_k}$$

k=0, 1 , . . . ,

Where $-H_k^{-1} . g_k$ is the Newton direction.

If the function has a minimum, and the second order sufficiency conditions for a minimum hold, then H is positive definite and, therefore, nonsingular at any point x.

If *f*(x) is a *n*-dimensional quadratic function, then the Newton method will converge in only one step from any starting point. If not, the Newton method will iterate incorporating a line search to calculate changes in x.

Based on the number of iterations the convergence may seem fast. However, each iteration includes the calculation of the second derivative and handling of the Hessian which can be very time-

consuming, especially for large systems. Another drawback of the Newton method is that it may not converge from any starting point.

Nevertheless, the Newton method is still considered a popular optimization technique for unconstrained nonlinear problems due to its fast quadratic convergence.

Constrained Gradient Methods

The gradient based algorithms described earlier are generally developed for unconstrained optimization problems. However, there are a certain class of algorithms that use the unconstrained optimization techniques to solve constrained problems. In the following I will discuss some methods that belong to that class.

Penalty-Methods

Penalty methods generally replace a constraint optimization problem by a sequence of unconstrained problems. The corresponding minimization problems are formed using a mathematical function that adds a penalty term to the objective function. The penalty function will increase the objective (for a minimization problem) depending on the value of the violated constraints but would remain constant otherwise.

Two major classes of Penalty methods exist depending on the formulation of the penalty function. The first class are known as *Exterior Penalty Methods* and use a sequence of infeasible points while adding a penalty for infeasibility. Feasibility is obtained only at the optimum. On the other hand, the second class of penalty methods adds a barrier to ensure that a feasible solution never becomes infeasible. These are referred to as *Interior Penalty Methods* or barrier function methods.

It is important to ensure that the penalty does not dominate the objective function during initial iterations of exterior point method. There are many methods to choose the penalty parameter sequence but the simplest is to keep it constant during all iterations.

Interior penalty methods have the advantage that if convergence is not achieved, a feasible solution is still maintained. Exterior penalty methods on the other hand have the advantage that they are less likely to be stuck in a local minimum. They are also more robust because in practice it is not always possible to have a feasible starting point. However, Exterior penalty functions typically require more function evaluations.

In general, these methods are sufficient for special purposes and provide easy techniques to consider constraints in an optimization problem using unconstrained optimization algorithms.

Sequential Quadratic Programming

Sequential Quadratic Programming (SQP) is one of the most popular and robust methods for solving nonlinearly constrained optimization problems. Like many other optimization methods, SQP is not a single unique algorithm, but rather a theoretical method from which several particular algorithms have evolved (Boggs and Tolle, 1995).

The basic method of SQP is analogous to Newton's method for unconstrained optimization in which an iterative approach is implemented to solving a series of subproblems that hopefully yield a step toward the problem solution.

The SQP models a nonlinear program at a given approximate solution X^k using a quadratic programming subproblem that substitutes the objective function with the quadratic approximation and replaces the constraint functions by linear approximations.

$$q_k(d) = \nabla f(X_k)^T d + \frac{1}{2} d^T \nabla_{xx}^2 \mathcal{L}(x_{k1}\lambda_k) d$$

This defines a search direction d_k as a solution to the quadratic programming subproblem. The solution to this subproblem is then used to construct a better approximation X^{k+1}. This process is iterated to create a sequence of approximations that hopefully will converge to a solution X^* (Boggs and Tolle, 1995). If the starting point x_0 is close to X^* and the Lagrange multiplier estimates $\{\lambda_k\}$ are close to λ^*, then the sequence generated converges to X^* at a second-order rate.

If the problem is unconstrained, then only the objective function is approximated, and the local model is quadratic, thus the SQP functions like the Newton's method. If the problem only includes equality constraints, then the method is similar to applying the Newton's method to the first-order optimality conditions or the Karush-Kuhn-Tucker conditions.

In SQP, the determination of the search direction based on solving quadratic subproblems requires considerably more computational effort than simple search methods. Based on the similarity with the Newton method it would be expected that the SQP method would share characteristics such as rapid convergence when the iterates

are close to the solution, the need for controlling the behavior when the iterates are far from a solution. However, the presence of constraints makes both the analysis and implementation of SQP methods considerably more complex (Boggs and Tolle, 1995).

It is worth noting that the SQP is not a feasible-point method and therefore neither the initial point nor any of the subsequent iterates need be feasible. This is an important feature in SQP since finding a feasible point with the existence nonlinear constraints may be nearly as hard as solving nonlinear program itself (Boggs and Tolle, 1995).

5.6.4.2 Heuristic Algorithms

Heuristic algorithms are a class of algorithms that are able to find an acceptable solution(s) to an optimization problem, but for which there is no formal mathematical proof of its correctness. This class of optimization algorithms has proved to be practical in many scenarios. In the following sections I will discuss a few of the better known algorithms of this class.

5.6.4.2.1 Evolutionary Algorithms

Evolutionary search algorithms are inspired by and based upon evolution in nature. They permit us to exploit the remarkable properties of natural evolution. These algorithms typically use an analogy with natural evolution to search by evolving solutions to problems. Instead of working with one solution at a time in the search space, these algorithms consider a large collection or population of solutions at once (Bentley, 1999).

There are four main types of evolutionary algorithms in use today, three of which were independently developed more than forty years ago, with the fourth being developed in the last couple of decades. These algorithms are:

Genetic algorithms
Genetic algorithms (GA) were developed by John Holland (University of Michigan in Ann Arbor) in the early 1960s, and made famous by David Goldberg (1989). Holland's original intention was to understand the principles of adaptive systems (Dumitrescu, 2000).

Evolutionary programming
Evolutionary programming (EP) was devised by Lawrence J. Fogel (1962) and developed further by his son David Fogel (1992), as an attempt to simulate intelligent behavior by means of finite-state machines (Dumitrescu, 2000; Corne and Bentley, 2002).

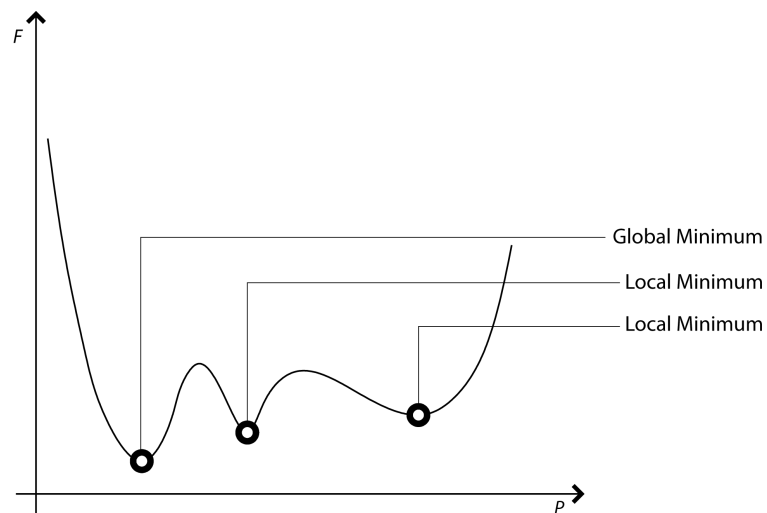
Evolution strategies
Evolution strategies (ES) (1965) originate in the work of Bienert, Rechenberg, and Schwefel concerning a method to optimize parameters for aerotechnology devices. Today this method is strongly promoted by Thomas Back (1996) (Dumitrescu, 2000; Corne and Bentley, 2002).

Genetic programming
Genetic Programming (GP) is a more recent and very popular development of John Koza (1989). The aim of genetic programming is to develop, in an automated way, computer programs for solving specific problems. Genetic programming is therefore a domain-independent approach to automatic programming (Dumitrescu, 2000)

The field of evolutionary computation has grown up around these techniques. Evolution-based algorithms have been found to be some of the most flexible, efficient, and robust of all search and optimization algorithms known to computer science (Goldberg, 1989). In the following section I will focus on discussing genetic algorithms.

Figure 5.45:

The difference between a local minimum and a global minimum



Genetic algorithms

Genetic algorithms are a particular class of evolutionary algorithms. They provide optimization and search techniques adequate for searching noisy solution spaces. GAs are categorized as global search heuristics because they search a population of solutions instead of a

single solution which limits the probability of getting trapped in a local minimum (figure 5.45).

GAs search by randomly sampling the solution space, and then use genetic operators to direct a hill-climbing process based on fitness (objective) function values (Goldberg 1989). GAs require a genetic representation of the solution as well as a fitness function to evaluate it.

GAs are implemented as abstract representations called chromosomes or genotypes of the candidate solutions, also called individuals or phenotypes. These genotypes are usually represented as binary strings, but other encodings are also possible.

A GA works by producing a group of solutions called a population. Each new population created is called a generation. GAs use genetic operators inspired by evolutionary biology such as selection, mutation and crossover.

Constraints are implemented through the use of penalty functions. If a solution does not meet constrains in the system, then a penalty is added to the fitness of the design solution according to the degree of violation.

The GA starts by producing a population of randomly generated individuals. This is carried out in successive generations. In each generation, the fitness of each individual in the population is assessed. Based on their fitness, a number of individuals are selected from the current population and modified using genetic operators to produce a new population with higher average fitness than the previous population. This new population is then used in the next iteration of the algorithm and the cycle continues till a termination criterion is reached.

Genetic Representation

The GA's representation is done at two levels, namely at the genotype level and the phenotype level. The genotype is the implicit representation of an individual design solution. The standard representation of the genotype is a sequence of coded instructions stored in an array of bits called a chromosome. The chromosome encodes the parameters of interest that are related to that individual. A chromosome is formed of alleles that represent the coding bits. All the genetic operations including crossover and mutation happen at the genotype level.

The standard genetic representation has a fixed size which facilitates simple crossover operations because parts of the genotype are easily aligned. Representations that utilize a variable length can also be used although crossover implementation becomes more difficult.

The phenotype, on the other hand, is the interpretation of genotype at the physical level. It is the external perceptible representation of the genotype. The behaviors of a design solution can be observed at this level. Therefore, the analysis task is performed to design solutions at this level.

Fitness function

A fitness function of any particular individual corresponds to the value of the objective function that measures the quality of a chromosome. It is then ranked against all the other chromosomes. The probability of selecting a specific solution for reproduction is proportional to the fitness of that solution.

Population Size

The GA produces a group of solutions called a population. The initial population is generated randomly and should cover the entire search space. Solutions may also be seeded in regions where good solutions maybe found.

The population size depends on the nature of the problem. A very small population would not contain enough diversity in the initial genetic pool to ensure that good solutions are found by the algorithm (Goldberg, 1989). Therefore, the initial population must be large enough to provide a diverse genetic pool that contains substantial information in the search space that would eventually lead to better convergence. However, a large population needs many generations to converge which would cause time penalties. Hence, a moderate-sized population may be a sensible compromise between finding good solutions and speed.

Genetic operators

Once we have the genetic representation, the fitness function and the population size defined, the GA can proceed by initializing a random population. Genetic operators control and improve the evolution of successive generations. The three basic genetic operators are selection, crossover and mutation.

-Selection

In each successive generation, a number of solutions from the current population are selected to breed a new generation. These solutions are selected on the basis of their fitness.

There are several selection methods. Some methods evaluate the fitness of each solution and then select the best solutions. Other methods only evaluate a sample of the population to minimize time expenditure.

In general, most methods are stochastic and allow a small proportion of less fit solutions to be selected. This facilitates the diversification of the population which then can help in preventing premature convergence.

Common selection methods include biased roulette wheel selection and tournament selection methods.

In the biased roulette wheel selection method, each current individual in the population has a roulette wheel slot sized in proportion to its fitness (Goldberg, 1989). This fitness is used to define a probability of selection to each individual. If f_x is the fitness of individual *x* in the population, Then its probability of being selected is:

$$P(x) = \frac{f(x)}{\sum_{j=1}^{n} f(j)}$$

where *n* is the number of individuals in the population.

Using this method, individuals with a higher fitness have a higher probability of being selected to the next generation. However, individuals with poor fitness also have a chance of being selected, albeit a smaller chance.

The tournament selection method chooses a random group of individuals from the population to run a tournament among. The winner is selected based on fitness. If the tournament group size is large, individuals with a poor fitness have a smaller chance of being selected.

Other strategies that can be implemented in conjunction with selection include elitism. Employing elitism in the GA implies passing the best solution from one generation to the next with the goal of preserving good genetic information contained in that solution.

- Crossover

Crossover is a genetic operator used to vary the encoding of chromosomes from one generation to the next. That is achieved by swapping parts of two randomly chosen chromosomes to create a new individual but not new genetic information. It is similar to biological crossover on which genetic algorithms are based.

Elite solutions do not go through crossover, but rather an exact copy of them is carried to the next generation. Many crossover operators exist, but the most common are the one-point crossover, two-point crossover and uniform crossover.

In a single crossover point, the algorithm identifies a crossover point on the chromosomes of the two parents selected for reproduction. The information left on either chromosomes is swapped between the two parents creating new offspring.

The two-point crossover is similar, but two crossover points are selected on the parent chromosomes. The rest of the information between the two points is swapped between the parents creating new offspring.

In the uniform crossover, bits in chromosomes of two parents are swapped with an equal probability of typically 0.5 to create new offspring.

- Mutation

Mutation in GAs is similar to biological mutation and is used to maintain genetic diversity from one generation to the next. The mutation operator involves a probability that a random allele in a chromosome will be changed from its original state to identify new points in the search space. Mutation is therefore an operator that acts locally.

A general method for applying the mutation operator is to generate a random variable for each allele in a chromosome. This variable will help determine whether or not a particular allele will be mutated. This mutation will introduce new genetic information that was not contained in the initial population.

The main reason for implementing mutation in GAs is to assist the algorithm in avoiding local optima by preventing it from generating populations of similar chromosomes that affect the evolution and result in premature convergence.

Termination

The GA will continue generating populations until a termination condition has been reached. This could be due to reaching the maximum number of generations, or a solution or set of solutions have been found with satisfactory fitness levels, or the highest fitness levels reached a plateau and successive iterations are not producing better results.

5.6.4.2.2 Simulated Annealing

Simulated annealing (SA) is another general heuristic technique for solving optimization problems. It incorporates randomization techniques and is often used for discrete search space.

SA may be more efficient than exhaustive enumeration for certain problems, especially if the intent is to identify an acceptable solution rather than the optimum solution.

The method was first introduced in 1983 by S. Kirkpatrick et al. (1983). Simulated annealing is based on the analogy between annealing in metallurgy and solving optimization problems. Annealing in metallurgy is a technique that involves both a heating and controlled cooling processes of a metal solid to reduce its crystals defects and increase their size.

Initially the metal solid is heated up and melted, causing the particles to move from their initial state of minimum internal energy into states of higher energy and rearrange themselves in the liquid phase.

This is followed by a slow lowering of the temperature which gives them a chance of finding configurations with lower internal energy. If the cooling is too fast and the solid does not reach thermal equilibrium for each temperature value, then defects get frozen into the solid producing metastable amorphous structures instead of the low-energy crystalline lattice structure (van Laarhoven, 1987).

By analogy, each point i of the search space corresponds to a state of some physical system, and the function $E(i)$ to be minimized corresponds to the internal energy of the system in that state. The aim is to move the system from an arbitrary initial state to a state with the minimum possible energy.

The SA algorithm functions like a sequence of Metropolis algorithms that are executed at decreasing values of the control parameter. Each step of the SA algorithm substitutes the current state by a

random neighbor state that is chosen based on a probability that depends on the difference between the corresponding function values and on a control parameter temperature *T*.

Algorithm

- The basic iteration

At each iteration, given a current state *i*, the algorithm generates a possible transition from that state to a neighbor *j*. For each state *i*, a neighborhood *N(i)* consists of all the states that can be reached from *i*.

If that neighbor has a lower cost, the current solution may be replaced by it. A probability decides between moving the system to the new state *j* or remaining at state *i*. The probabilities are selected so that the system eventually moves to a state of lower energy. This step is iterated until an acceptable state (solution) is reached or the algorithm is terminated due to the exhaustion of computational resources or time restrictions.

- Acceptance probability

The acceptance probability *P* indicates the probability of accepting the candidate state *j.* The function $P(e,e',T)$ depends on the energies of the two states, $e = E(i)$ and $e' = E(j)$, as well as the temperature *T* which is a time-varying control parameter.

The new state *j* is accepted with a probability of 1 when $e' < e$ which implies a move downhill. If $e' > e$ the new state *j* can still be accepted, although that implies that the method moves to a worse state with higher energy. This is an important feature that prevents the method from being locked in a local minimum.

As the difference $(e' - e)$ increases the probability of accepting a move decreases which makes large uphill moves less likely. As the control parameter *T* goes to zero, and if $e' < e$, then the probability *P* goes to a positive value or to zero if $e' > e$. Therefore for small T values the method will prefer moves that go downhill to lower energy values. This process continues at each value of the control parameter *T* until equilibrium is reached.

- The cooling schedule

Initially *T* is set to a high value and is gradually decreased (cooled) at each iteration according to an annealing schedule. There are two

kinds of cooling schedules: static and dynamic (Aarts et al., 1997). In the static cooling schedule, the parameters remain unchanged through the implementation of the algorithm. On the other hand, the parameters in the dynamic cooling schedule are changed adaptively during the implementation of the algorithm.

It has been demonstrated theoretically that for any given finite problem, the probability that the simulated annealing algorithm terminates with the global optimal solution approaches 1 as the annealing schedule is extended (Granville et al., 1994).

In practice the algorithm can be terminated when a state is obtained whose objective function value is no worse than any of its neighbors (van Laarhoven, 1987) or if the value of the objective function remains unchanged for a number of consecutive trials.

5.6.4.2.3 Tabu Search

Tabu search is a heuristic algorithm that searches local neighborhoods for the best possible path to progress (Hertz et aI., 1997). It was initially introduced by Fred Glover (Glover, 1989).

Tabu search utilizes a neighborhood search approach to iteratively move from a solution i to a solution j in the neighborhood $N(i)$, until some termination criterion has been satisfied.

Unlike hill-climbing methods, which can easily be trapped in local minima, Tabu search continues exploration picking the best available move at each step even if it is a non-improving one. This presents the risk of visiting once more a solution that has already been evaluated which could generate cycles within the search process. Therefore, when a solution has been identified it is marked as tabu (taboo) so that the algorithm does not visit that solution again.

The tabu search algorithm adjusts the neighborhood structure of each solution as the search advances to explore the unexplored regions in the search space. The new neighborhood solutions allowed in $N(i)$ are determined by the use of memory structures.

Tabu search makes use of a form of short-term memory to maintain information on the journey through the final solutions visited. This information is stored in a list of size T called a tabu list and determines the solutions admitted to $N(i)$. It includes solutions that have been visited in the recent past and excludes solutions already in the tabu list. Therefore, the structure of the neighborhood $N(i)$ will thus depend on the journey followed to reach i.

Due to large memory use, it may be unfeasible to store lists of solutions, hence Tabu Search uses a list of moves instead and often stores only a portion of the attributes required to describe a move or the solution to which it is applied (Glover, 1989). This portion of the attributes may potentially be shared by other moves or solutions. This might prohibit not yet visited solutions that have certain attributes. Some excellent solutions might now be avoided. To solve this problem aspiration criteria are implemented. An example of an aspiration criterion is to keep solutions that are better than the current best solution. This way, aspiration criteria can include otherwise-excluded solutions.

In addition to short-term memory, tabu Search also uses intermediate and long-term memory structures. Intermediate memory is used to carry out temporary intensification of the search around a certain area of the solution space that contains a good solution. This is done by storing and comparing attributes from current best solutions. Common attributes are considered when searching for new solutions. Long-term memory is implemented to provide a diversification process of the search. The diversification process guides the search to regions in contrast to ones examined by penalizing attributes that are found to be common in previous executions of the search process (Glover, 1989). This provides the ability to learn from previous steps and solutions in the search journey.

One apparent feature in tabu Search is that it functions as a greedy algorithm. According to Glover (1989) this is based on two considerations: firstly, that many optimization problems can be solved optimally by making the best move at each step; secondly, that local optimality does not represent an obstacle for Tabu Search because of its procedural organization and its use of short-term memory.

6. Integration

6.1 What is Integration?

The term "integration", similar to the term "system", although widely used, has a lot of connotations which may carry some ambiguity for the listener or reader. The Webster online dictionary defines integration as the "act of combining parts into an integral whole". The key notion of integration is the assembly and combination of individual parts and design components into a "system" that collectively satisfies all the functional and operational requirements which would not be achieved by its subsets alone (Grady, 1994).

Decomposition and formulation usually lie at the front end of the MDDS development process, while integration lies at the tail end. The design modeling process lies in the middle, where the synthesis, analysis, evaluation and optimization activities occur. Integration is the materialization of the formulation and modeling processes. If the formulation stage is where an architecture plan is devised, then integration is where the final stages of that plan are executed. Integration within the MDDS context deals with connecting the different modules into one coherent software.

Like formulation, integration can occur at different levels, especially in the development of the system. Achieving a system in final product form, as highlighted by Eppinger (1997), requires integration at various levels, such as the integration of components into sub-systems, sub-systems into systems, and systems into products. Integration works mainly on unifying product components and process components into one whole. The success of integration and subsequent evaluation lies, however, in the correctness, precision and coordination of engineering activities at each level.

System integration efforts by design teams are usually complex and rely heavily on technical expertise and advanced planning and preparation in the formulation stage. These design teams usually work on an evolving, complex problem, through its sub-systems and components, on two basic levels: within and across teams. The cross-

functional team approach, a common practice of concurrent engineering (Eppinger, 1997), involves the simultaneous efforts of addressing design and production. Here the interest in sub-systems and components of complex systems exists at the intersection of multiple teams, disciplines and technologies, in order to satisfy solutions for complex problems. This necessity originates from basic principles of system engineering, human psychology, and human existence (Grady, 1994). Although human limitations seem to drive integration, the issue of combining the work of multiple teams from varying functional disciplines is not a simple one, as it also entails working in several process steps over time on a variety of product system components.

In this chapter, we are interested in integrating components to produce a software product or system that can generate design solutions of physical artifacts. Our system will include subsystems that must interface with other subsystems in order to exchange information. The components of each subsystem must interact in the same fashion. Our focus here will be on the integration of design systems from the informational perspective, rather than the physical perspective. Since the design system is computational, we will focus on the integration of information and computational systems.

6.2 Interface Design

An interface is defined as the place where communication of information or activities is facilitated between the different components and modules of a specific system (Grady, 1994).

Baldwin and Clark (2000) define an interface as a pre-established way to resolve potential conflicts between interacting parts of a design. It is like a treaty between two or more subelements. To minimize conflict, the terms of these treaties—the detailed interface specifications—need to be set in advance and known to the affected parties. Thus interfaces are part of a common information set that those working on the design need to assimilate. Interfaces are visible information.

In general, interfaces are completed between components through interface media, such as electrical or radio signals, physical contact, flow of fluids, etc. Interface media are usually provided by the system environment or a component of the system architecture. The interface however is represented in the functionality facilitated by the media rather than the media itself.

An interface can represent both distinct worlds discussed earlier, the

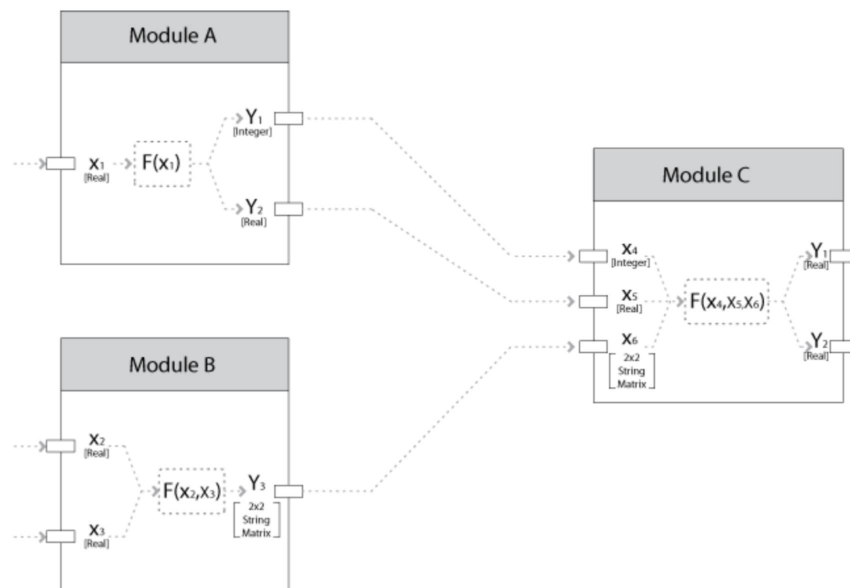
physical and the informational. Physical interfaces are usually associated with the physical form of the component fitting or matching, such as the mounting bolts in a column beam connection. Function can also be represented through the flow of forces, such as the load bearing function flowing from beams to columns.

In the software world, a similar matching or compatibility must occur between input and output data of the system modules. Compatibility here refers to the correct and meaningful way by which two activities interact semantically (Gao et al., 2003).

It is known that the system functionality of large complex problems is decomposed into smaller system elements and smaller problems. These problems have interfaces between them that must have compatible representations on both terminals (figure 6.1). Hence, every module defines the specifications and requirements for its subordinate modules. At the same time, these subordinate modules are precisely constructed and specified according to these requirements.

Figure 6.1:

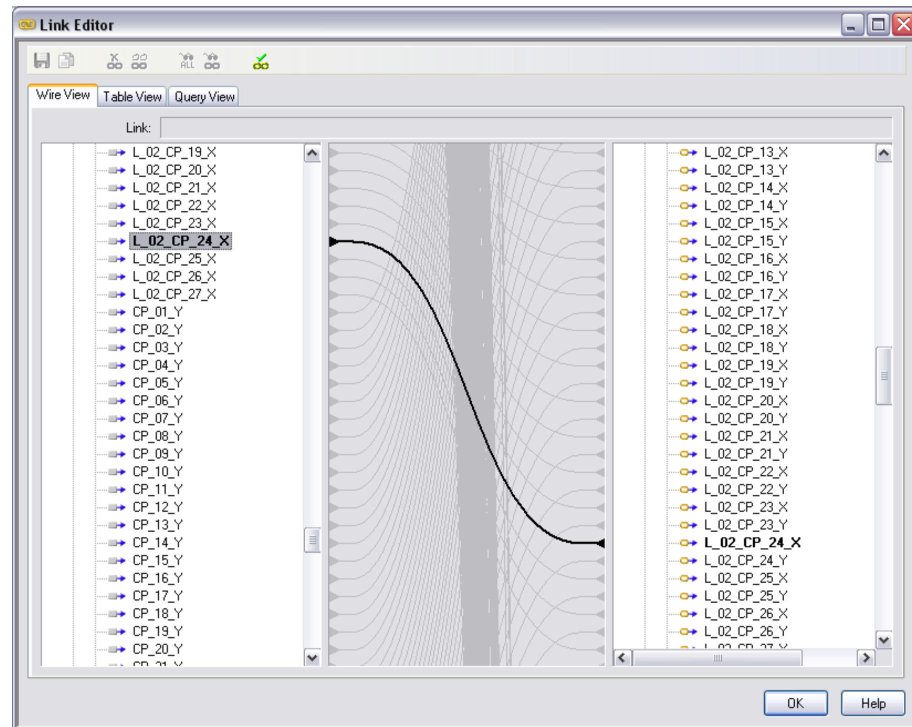
Interfaces between different modules have to be compatible.



The major complexity occurs, however, when different disciplines or team specialists are responsible for the integration at these opposite terminals of the interface. The greatest effort in this cross product integration lies in trying to manage the matching process between interface terminals and checking data compatibility at both ends (figure 6.2).

Figure 6.2:

Matching interfaces between modules with many variables can be a difficult task.



In order to arrive at an optimized system architecture and interface, the need arises to accurately and clearly define the location of interface planes in the system, the responsibility of disciplines and specialists, and the congruent understanding of concept requirements and specifications by both media and specialists. This process is often referred to as interface analysis (Grady, 1994).

The methods proposed previously in formulation can be used to define system interfaces through a combination of schematic block diagrams, integration models, DSM/N2 diagrams and interface dictionaries. Changes may occur in the system architecture during this process due to allocation of system functionality.

However, the principal technique for resolving complexity of interface compatibility, whether in a physical or information system, lies in minimizing the number of interfaces. In general, this can be achieved by maximizing the capability of system component interaction while minimizing the need for the system to interact. At the same time, components of the system architecture can be aggregated in order to reduce the need for cross-organizational and cross-discipline interfaces.

6.3 Module Integration Modes

Software system integration involves the combination of individually tested software components into an integrated whole. This occurs when the components are combined into subsystems or when subsystems are combined into products (Software Engineering Institute, 2008). The field of application integration comes in many forms, whether it describes the integration between components of a single software system or the integration between different systems.

Several technologies have been developed to address integration. One of these technologies is component-based software engineering (CBSE) which emerged as a branch of software engineering which involves the decomposition of systems into functional components that have well-defined interfaces used for component integration.
In CBSE, components are basically objects that are written to a specification and adhere to it. They are of a higher level of abstraction than objects; they do not share state and communicate by means of data exchange.

When software is considered as a component, it describes a system element that offers a predefined service or event, and is able to communicate with other components. Messerschmitt and Szyperski (2003) describe some fundamental characteristics for a software component. In their definition, a software component is a unit of independent deployment and versioning that is non context-specific, encapsulated, of multiple use, and composable with other components.

The essence of integration lies primarily in careful management of component interfaces. According to component interfaces, this involves the assumptions that the programmers of each component can safely make about the other component. Integration is assumed to progress smoothly between components if these interfaces are well-defined and carefully documented (Parnas, 1972).

Software components usually take the form of objects or groups of objects, in the object-oriented programming context, that hold on to an interface language of some sort. This requires that all the information and assumptions about the component behavior, its consumed resources, its response in reaction to errors, and the mechanism of connection and interaction with other components are all well-defined, taken into consideration and evaluated in planning and budgeting phases.

Reusability is a significant characteristic of a high-quality software component when it comes to accessing or sharing components across execution contexts or network links. The component should be designed and executed in a way that enables its reuse in many different programs.

In the 1960s scientific subroutine libraries were constructed and were reusable in a wide range of engineering and scientific applications. However, these libraries had a limited domain of application although they reused well-defined algorithms effectively. Modern reusable components work on encapsulating both the data structures and the algorithms that are applied to them
In order for a software component to be effectively reusable, there has to be a lot of effort exerted in writing the component. It needs to be fully documented and constructed while considering that it will actually be exposed to unpredicted uses.

It should be noted that the literature uses both terms, components and modules, usually to describe the same thing. The main difference between them is usually in the reuse scope of each. Because a software module is developed for a specific project it tends to have a narrow reuse scope. A modern software component, on the other hand, provides multiple-level granularities for reuse on a large scope (Gao et al., 2003). In this thesis I will be using both terms interchangeably. The mathematical models described in the previous chapter can represent such a module or component.

There are different approaches to achieve integration between system components. These include having all the applications totally integrated in one software, or using middleware, such as distributed technologies, or applying technologies, such as wrapping which has been widely used in many problem-solving environments. Another approach involves using web services to provide for the data integration between different components.

6.3.1 Middleware

Middleware is essentially a piece of computer software that connects software components or applications for the purpose of data exchange. Middleware became popular as a solution to the problem of linking newer applications to older legacy systems. It also permits distributed processing, where multiple processes and applications that run on one or several machines are connected to create a larger application and interact together across a network (Software Engineering Institute, 2008). Middleware is defined by Object Web as "the software layer that lies between the operating

system and applications on each side of a distributed computing system in a network" (Object web open source Middleware, 2008).

Middleware allows users to share distributed resources such as applications, data, computers, and networks. It supports effective collaboration and communication tools. Middleware is also vital for Internet computing, and high-performance parallel computing. It also provides working architectures and approaches that can be extended to the larger set of Internet and network users (Sun and Blatecky, 2004).

Middleware provides a functional set of application programming interfaces and uses a particular class of software products that act as intermediaries between user interfaces on one hand and data generators and repositories on the other. In general, it comprises a library of functions and allows multiple applications to communicate to those functions from the common library instead of regenerating them for each application.

Middleware allows applications to be independent from network services. It also makes applications reliable and always available when compared to the operating system and network services. At the same time, it makes the applications locate transparently across the network. This facilitates the interaction mechanism with other services or applications and provides consistency, security, privacy, and capabilities (Sun and Blatecky, 2004). Middleware technology supports the shift to interoperability and coherent distributed architectures and includes web servers, transaction monitors, and messaging-and-queuing software.

In general, middleware is used for distributed computing, distributed technologies and distributed application frameworks that have been used to build complex services. Types of middleware typically include middleware between applications and database servers, such as SQL-oriented Data Access, and application servers which are pieces of software that run multiple software components and facilitate the running of other applications. Combining application servers and software components is usually known as distributed computing.

Examples of distributed technologies include Enterprise JavaBeans from Sun Microsystems, the Java specific EJB (Enterprise Java Beans), distributed computing software components, such as .NET Remoting from Microsoft, Web Services, XML-RPC, the predecessor of SOAP, CORBA and the CORBA Component Model from the Object Management Group, and CORBA (Common Object Request Broker Architecture) which is both platform and language independent.

CORBA has specifically been successful as a distributed component framework in many areas including telecommunications, finance, e-commerce, and healthcare. That is why it is worth discussing here.

CORBA is an open standard for distributed object computing defined by the Object Management Group (OMG), a not-for-profit consortium. Software components written in multiple computer languages and running on multiple computers and operating systems can use the vendor-independent architecture and infrastructure of CORBA to call on each other's services and work together over networks. As an object bus, it allows clients to invoke methods on remote objects at the server independent of their location and the language they are originally written in.

The CORBA specification specifies an object request broker (ORB) by which the application interacts with other objects. ORB mediates the interaction between client and server on both the client and server sides, where the communication typically takes place via the Internet Inter-ORB Protocol (IIOP). CORBA objects in this case can either be collocated with the client or distributed on a remote server without having any effect on their implementation or use. ORBs take care of the details of this process.

The Interface Definition Language (IDL) defines the capabilities of CORBA objects, that is its operations or methods. These operations can take in input parameters and return values corresponding to some CORBA data-types and can also raise exceptions. CORBA uses the IDL to specify the interfaces that its objects will introduce to the outside world. It then specifies a mapping scheme from IDL to a specific implementation language such as C++ or Java. (Software Engineering Institute, 2008).

Some programming languages, e.g. Java, allow users to define a compilable specification separate from the body, where keeping a continuously integrated system using full specifications was found to be time and cost efficient for integration purposes. Although these languages are especially useful in catching integration bugs early on, they do not allow the specification of the full semantic interfaces of components (Software Engineering Institute, 2008). IDLs, on the other hand, describe an interface that facilitates communication between software components that do not essentially share a language. They offer a language-neutral bridge between two different systems, whether these systems use different operating systems or computer languages, which is typical in remote procedure call software.

Although the IDL interface definition is independent of any programming language, it has mappings through OMG standards to all popular languages like C, C++, Java, COBOL, Smalltalk, Ada, Lisp, Python, and IDLscript. This capacity to enable interoperability and separate interface from implementation is facilitated by OMG IDL and constitutes the essence of CORBA.

While the interface to each object is defined very explicitly, its implementation, running code and data are hidden from the rest of the system, or in other words *encapsulated* behind a boundary that clients may not cross. Clients thus access objects only through their advertised interface. They can only call the operations that the object exposes through its IDL interface. In addition, they can only address the input and output parameters that are included in those call specifications (Object Management Group, 2008).

Not only does CORBA provide users with a language and a platform-neutral remote procedure call specification, but also it defines services that are commonly required such as transactions and security, events, time, as well as some other domain-specific interface models. Moreover it is designed to be operating system-independent, so it can run on many platforms such as Win32, UNIX and real-time embedded systems.

6.3.1.1 Encapsulation (Wrappers)

Software encapsulation is based on the technology of wrapping. Thomas Dietrich of IBM first introduced the concept of the "wrapper" at the 00PSLA Conference in 1988 as the solution for existing legacy software in a new object oriented architecture (Dietrich, 1989).

One of the key properties of wrappers is that they are generic. Phoenix Integration (2007) defines a wrapper as a set of instructions that describe inputs, output, and how to execute an analysis. According to the property of generality, wrappers for a specific component should work generically for any component of the same type (e.g. A wrapper for Excel should work for any Excel document). At the same time, wrappers should provide functionality such that data can be input into that component, and any data type could be extracted.

According to Mowbray and Zahari's (1994), an object wrapper provides access to a legacy system through what is called an encapsulation layer. This encapsulation exposes only the properties and operations desired by the software architect. Mowbray and

Zahari describe seven basic techniques for wrapper implementation: remote procedure calls, file transfers, sockets or docking, application program interfaces, script procedures, macros and common headers. These techniques can be implemented individually or with each other to build connections between a service requester and a service provider (Mowbray and Zahari, 1994).

Ian Graham defines a wrapper as a software controller layer that allows object-oriented programs to access conventional programs as if they were objects (Sneed, 2000). According to Graham's Semantic Object Modeling Architecture (SOMA), the implementation of wrapping existing software components is significant, but it is not that easy. The wrapper software has to adapt incoming requests to wrapped software interfaces in a dynamic fashion due to data type incompatibility.

According to Seacord (2001) wrapping is a technique for integrating components whose interfaces cannot be controlled. These components include ones that are mined or acquired by means of a third party. It involves writing software that works as a mediator between the expected interface and the interface that the used component comes with. In pure wrapping, there is no alteration in the component; instead a new thin layer of software is introduced between the original component and its clients (Software Engineering Institute, 2008). This layer provides the new interface by translating to and from the original component.

One of the advantages of wrapping is the concept of reusing existing assets with little or no internal modification. Wrapping inhibits the ripple effect that occurs when any modification takes place, thus preventing the influences on other associated software that happen due to documentation and test case changes. Instead, wrapping a kind of an "as-is" reuse of many of the component associated assets, such as its test cases and internal design documentation (Phoenix Integration, 2007).

Wrapping is also seen as a substitute strategy to reengineering and redevelopment in the context of encapsulating existing legacy software for reuse in new distributed architectures (Sneed, 2000), as it is lower in cost and has lower risks than conventional reengineering. Application wrappers encapsulate batch processes or online transactions. Legacy components are considered by new client applications as objects. These objects are invoked to perform specific tasks such as producing reports. Function wrappers provide interfaces to call individual functions within wrapped programs. There is only limited access however, as only specific parts can be

called and not the whole program.

Wrapping legacy software is usually done in three basic steps: first, the wrapper should be constructed; next, the target programs should be adapted; finally, testing should occur to validate the interaction between the wrapper and the target programs. In general, a wrapper uses message-passing mechanisms to connect to its clients. As input, it receives incoming requests. It then reformats them, loads the wrapped object and calls upon it using the reformatted arguments. Concerning output, the wrapper obtains the results from the wrapped object, reformats and sends them back accordingly to the original requester.

The way a wrapper works is described as follows. Being a shell between middle software and user software, the wrapper first receives messages from the client application. It translates these messages into an internal format and then calls upon the target software. It also converts the target software outputs to an external format. Finally, it sends the outputs back to the client application.

There are still no automatic wrapper generators for legacy codes that can operate at different degrees of granularity, or that can wrap the entire code or code sub-routines automatically. Some progress has been made though towards achieving this goal. The lack of these types of wrappers primarily owes to the fact that current tools, such as Fortran and Java translators, cannot manage the specialized data types appropriately and are insufficient for translating large application codes (Li et al. 2004).

6.3.1.2 Web Services

Web services represent an emerging distributed middleware technology. They employ a simple XML-based protocol to enable data exchange between applications across the Web. Services here are described in terms of the accepted and created messages. Users of these services do not need to have any knowledge of the object model, programming language or other details of the implementation. The only thing they need to do is to be capable of sending and receiving messages.

SOAP (simple object access protocol) lies at the core of web services. SOAP is an XML based communication protocol for interacting with Web services. Specifications or interfaces of the services can be described using WSDL (web services description language). WSDL is an XML-based general framework that describes network services as groups of communication endpoints that can exchange messages. It

identifies the location of a service, what operations are supported, as well as the format of the messages that are to be exchanged according to the way the service is called upon.

6.3.1.2.1 EXtensible Markup Language (XML)

EXtensible Markup Language (XML) is a task and schema specification and a set of rules that provide standard ways to define processes and information and design text formats for information structuring (Harrison et al., 2004). XML is a standards-based protocol that can be used as a means of communication between software components. Many integration opportunities can be realized, whether the integration occurs between components of a single software system or between systems. XML can simply be considered as a mark-up language for annotating text documents. It specifies how tags, which are denoted by angle brackets, are used to organize written information.

XML provides an improvement over binary or textual information, as it describes a hierarchical relationship between all data elements. Parsers in XML can be written using standards like DOM or SAX in modern languages such as C++ and Java (Harrison et al., 2004). Since XML is platform-independent, the integration of external applications becomes less dependent on the software platform in which the applications are executed. This integration can be done using an XML derivative, such as SOAP or any standard industry-based XML extensions. An integrated solution can be realized through the connectivity that can be implemented using messaging queuing technologies, such as those available through IBM, Microsoft and other vendors.

Potential for component reusability has been greatly improved with the growing number of developed XML-based standards. The numerous published XML formats for encoded data allow for the development of cross-system code. The components that process format-specific XML can be reused once generated to process any given data type.

As XML reduces data manipulation and delivery time, most database vendors use it to provide interfaces to their engines. At the same time, most database servers support data access using XML. XML also works well as a data formatting method for passing data across the Internet. This is extremely beneficial for users developing components in existing systems. The process of adding web interfaces to application components can be simplified according to the number of available XML integration packages.

A wide range of data specific formats is available to allow interoperability, and several tools and applications currently implement import/export options to accept and write out XML. In order to facilitate automated access to complex services, some companies, led by Microsoft and IBM and being handled recently by the XML Protocol Activity group under W3C, have standardized on SOAP as a lightweight protocol that is based on XML to exchange messages over the Web. It is worth noting that XML on its own is not considered middleware, but SOAP, as a middleware specification, makes use of it.

6.3.1.2.2 SOAP

As mentioned earlier, SOAP is an emerging distributed middleware technology. It employs a lightweight and simple XML-based protocol to enable applications to support the exchange of structured and typed information across the Web based on a shared, decentralized, and open web infrastructure.
SOAP applications can be written in a variety of programming languages including Java, C++, C, Perl, and C#. These languages are used together with a multitude of Internet protocols and formats such as HTTP, SMTP, and MIME. Together they can support many applications, ranging from messaging systems to RPC (remote procedure calls). Any SOAP architecture consists of three basic parts: an envelope that describes the contents of a message and how to process it; a set of encoding rules to express instances of application-defined datatypes; and a convention to represent remote procedure calls and responses (Schmidt, 2001).

SOAP is therefore similar to CORBA's IIOP as it is a protocol whose purpose is to convey messages between applications (Schmidt, 2001). However, one of the main distinctions between CORBA and web service technologies like SOAP is that CORBA provides real object-oriented component architecture. Web services on the other hand are message based and not object-based. Also, there is a tight coupling between clients and servers in CORBA. They must both share the same interface, with a stub on the client-side and the corresponding skeleton on the server-side. Intermediation is not required in the direct interaction between client and server, except from the ORB which runs at both ends.

Everything is decoupled, however, in web services. The client sends and receives a message, while the response does not give immediate access to the next step. Web services are thus evolving into a role characterized by the integration of middleware rather than

competing with existing middleware technologies. It involves more middleware integration than middleware replacement. It is important to identify what this integration looks like, since middleware, in its own right, is seen as an integration technology. This has to do with the issue of application choreography but at a business process level (Schmidt, 2001).

In web services, integration seems to be moving toward a level of granularity more coarse-grained than typical CORBA-based integration and toward more loosely coupled systems. This loose coupling is achieved by minimizing interface dependencies and paying more attention to the exchange of XML-defined data, and so objects are considered more document-oriented than method-oriented (Schmidt, 2001).

Although CORBA sends information across networks, as opposed to the mere description of data in the case of XML on which most web services depend, this information exchange and system integration takes place within controlled environments (e.g. within intranets owned by a single company). Web services however are promising for integration both on the level of intranet and across the Internet (Schmidt, 2001).

This does not imply however that all other middleware technologies, including CORBA, are going to disappear. CORBA has already been shown capable of solving many distributed computing and integration problems. It has been able to provide solutions to problems that were insolvable otherwise through its high performance, dependability, scalability, and great flexibility. Other middleware technologies still have their places also, including .NET, J2EE, and EAI. (Schmidt, 2001). Further research is required to extend these potentials in the context of engineering design.

6.3.2 Integrated Computing Environments

Many key players participate in the design of any product. One of the important factors leading to success in product design is involving all these players early on in the product life cycle (Eppinger, 1995).

Each of the players may have computer models in their own tools of preference or in multiple software applications, including applications for math analysis, CAD systems, databases, spreadsheets and others. While all the built models represent a single product, there is no connectivity between the tools, thus requiring integration between them for facilitating data transfer. Based on the module integration modes discussed in the

previous section and other modes of integration, a variety of approaches and system architectures have been carried out to provide the required connectivity between the different design tools. We illustrate some below.

6.3.2.1 The One-Software Approach

This approach solves the connectivity between models by designing the functionality of many tools and embedding them into a single giant software or as extensions to it. Solving the connectivity and integration is thus less of a problem for the design team, since the software vendor solved these problems and provided the team with a set of tools within one package. Examples include CAD systems like CATIA, Unigraphics, and SolidWorks that integrate the functionality of spreadsheets finite element analysis, and computational fluid dynamics in one tool.

There are some considerable disadvantages, however, to this approach. The low level of comfort associated with being forced to use only one tool may cause engineers to be less productive. All the investment in the tools that would no longer be used is also lost. In terms of the integrated packages themselves, adding a lot of functionality to any piece of software does not necessarily enhance it. On the contrary, it could most probably make it less stable and less user friendly.

6.3.2.2 Problem Solving Environments

A Problem Solving Environment (PSE), as defined by Gallopoulos et al. (1994), is a complete, integrated computing environment for composing, compiling, and running applications in a specific area. This computer software aims at solving one class of problems through an easy to use interface (GUI) that is oriented primarily towards specialists in fields other than computer science.

PSEs were first introduced in the 1990s. For some years they were available for some specific domains. It was only in recent years that multi-disciplinary PSEs (M-PSEs) became widespread. Examples of M-PSEs include different kinds of Process Integration and Design Optimization (PIDO) Software. PSEs also exist as extensions to scientific programs like Matlab, Maple, and Mathematics (Li et al. 2004).

The main focus in PSEs lies in the ability to remotely use existing software in addition to reusing existing software libraries such as mathematical and visualization routines. Grid Computing is one of

the most significant fields in which PSEs are specifically used, where scientists and engineers at remote sites can interact through PSEs using standard software interfaces. Through this interaction, especially using distributed object technologies, such as CORBA and Java, the productivity of scientists and engineers is greatly enhanced.

Current work in PSEs has generally focused on building application specific PSEs. In general, a PSE must contain application development tools that allow end users to construct new applications or integrate libraries from other current applications in a way that makes it easier for users to extend within their domain. It must also contain development tools that facilitate the application implementation on a set of resources. Components (modules) can exist in different languages, locations, or platforms. They can be either created from scratch or wrapped from legacy codes.

A component is a self-contained program which has an interface that defines how it can be called upon by means of another component in addition to identifying the returned results upon operation completion. An interface in this context thus defines the different datatypes and return types associated with the component in addition to an execution model that describes all the libraries that should be taken into consideration to enable execution of the component (Li et al. 2004).

Through component interfaces, users can search for components that are appropriate for a specific application. The system components can be configured on instantiation, registered with event listeners and shared between repositories. Components are then connected together into data flow graphs and sent to a resource manager, where the application is executed on a workstation cluster or a combination of workstations and high performance machines.

Components can access a variety of available services according to the involved execution environment, such as an event service, a naming service, or a security service. Event services enable the coordination of activities between modules. Naming services facilitate the location of other modules. Security services verify module access, making sure that access to a specific module is only made from an authenticated module owner.

An important implementation technology for PSE infrastructure is component-based development, which allows wrapping existing scientific codes as components instead of rewriting them (Li et al.

2004).

The PSE infrastructure must support two types of users: application scientists or engineers who use PSEs to solve a specific problem, and programmers and software vendors who help accomplish the objectives of those scientists by developing components. PSE infrastructure should also allow the integration of third party products and application specific libraries.

PSEs should also support visual applications and web-based task submission. They must benefit from industry standards such as middleware (e.g. CORBA) and document tagging (XML). PSEs must include both resource management tools and application construction tools in an integrated fashion in order to run and schedule the constructed applications in an efficient manner. The tools required to build the required applications should be mostly domain independent. Most PSEs, however, do not actually provide an intuitive way to construct scientific applications through plugging software components together.

According to Li et al. (2004) some parts of the PSE should be considered domain independent and may be used for constructing applications in different domains, such as the Visual Programming Composition Environment (VPCE). Other parts are domain specific, where rules support particular types of components (Shields et al., 2004). The VPCE is a component repository that serves as a user interface for a PSE. The user can select a set of in-house components and combine them using a graphical composition area in the interface. The VPCE uses Java and CORBA to provide tools that allow building scientific applications from components. These tools ease the process of configuring components, integrating legacy codes into components and the process of designing and building new components.

There are current PSE projects that use component models, but most of them do not provide wrapping of existing scientific codes. They focus instead on creating data flow environments or on enabling users to write their own modules.

The advantage of the PSE approach is that it connects the tools that engineers are comfortable with in a generic way such that these connections can be managed by any Windows user (Wallace et al, 2000). It should allow engineers to continue using their preferred tools while easing the process of communication between those tools. On the software side, the only thing required from the new software is to facilitate the connectivity of legacy software.

Process Integration and Design Optimization (PIDO) Software

Process Integration and Design Optimization (PIDO) is an emerging line of software products. It aims at enabling users to integrate processes that use multiple digital design and analysis tools (Software Engineering Institute, 2008). These products allow the "wrapping" of software tools and legacy codes, in addition to publishing them on a computing network.

Through the graphical environment enabled by PIDO tools, users can generate an integrated MDA model by choosing published components and graphically linking their inputs and outputs. Through this approach, all disciplines can keep ownership of their codes. They can easily maintain and upgrade their codes as well as serving them from desired computing platforms. Engineers therefore do not have to learn new software. Examples include ModelCenter, AnalysisServer, DOME & Oculus Technologies, ISight, and Esteco. Since I will be using ModelCenter heavily in the thesis I will discuss it briefly in the following section.

ModelCenter

Phoenix Integration (2007) define ModelCenter as a tool that helps engineers design and analyze systems through automating multiple common computing tasks. The goal of using the software is to increase the efficiency of the design process by automating and simplifying these computing tasks. It saves engineering time and makes the design process less error prone. Multiple programs are connected together to form systems engineering models. Trade studies are performed on the models, and the results from multiple studies can be archived into a single project.

The first step in the process of using ModelCenter is wrapping a program on the Analysis Server. This program can be either an in-house code that uses input and output files, a commercial finite element program, or a Microsoft Excel spreadsheet. The Analysis Server is Java-based and runs on many platforms, allowing any analysis program to be wrapped and run on its own platform without further modification. The wrapped program produced by the Analysis Server is known as a component in ModelCenter, where the different components are accessible from other networked computers. A Model is thus a set of integrated components. Other ways to create components include the Script Component and Common Components.

ModelCenter graphically builds a model after the wrapping process is

complete. The model construction process involves selecting components in the Server Browser followed by dragging and dropping them into the Analysis View which provides a system-level model view. Instant relationships can be defined between the wrapped modules as soon as they are placed in the "work bench" environment. Attributes from a CAD module, for example, can be linked through ModelCenter's link editor to the corresponding parameters in a Matlab module or cells in an Excel spreadsheet module. Components are displayed as icons, while links are displayed as lines between the components (figure 6.3). Users can create, edit, and view links using the Link Editor. Viewing and editing model values can be done using the Component Tree which displays the model and all of its components and variables in a hierarchical fashion. Variables are displayed differently according to the variable types and states (Phoenix Integration 2007).

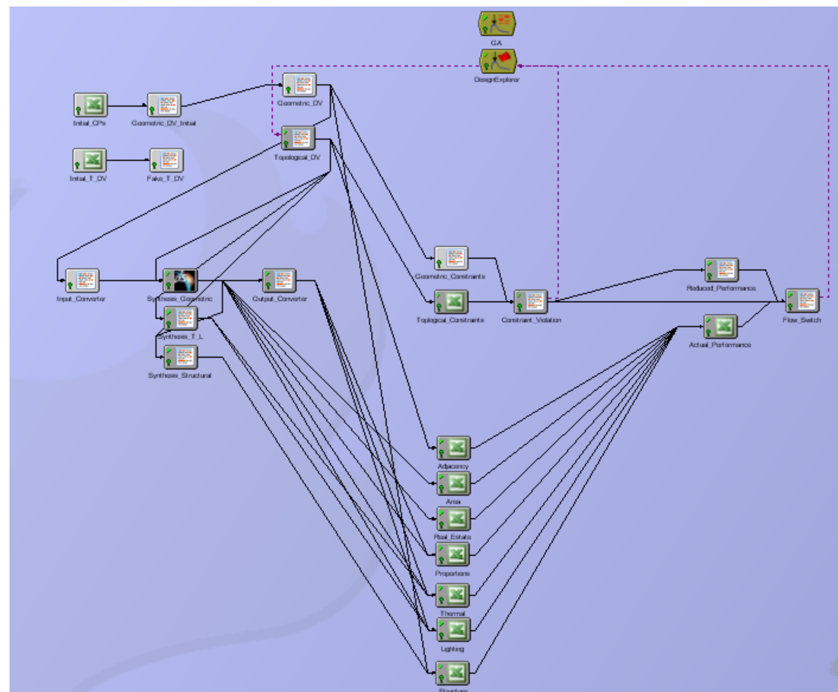


Figure 6.3:

In ModelCenter, components are displayed as icons while links are displayed as lines between the components.

Once alterations and iterations are carried out on one module, they are instantly propagated to other modules; this is an attribute that is promising for optimization. ModelCenter uses sophisticated algorithms to track different changes and impacts of variables. ModelCenter then uses a scheduler to decide which components to run and in what exact sequence (Phoenix Integration 2007).

7. Exploration

7.1. What is Exploration?

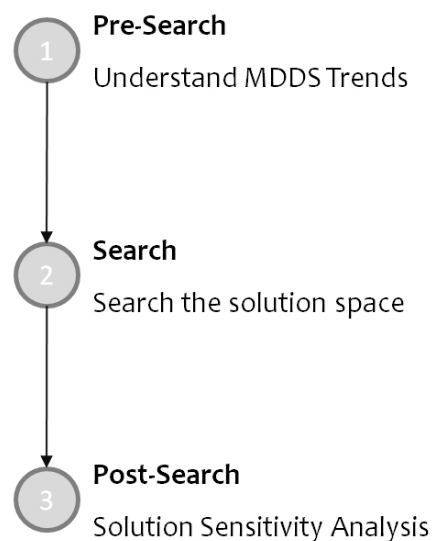
Changes in the design variables of one part of the system are rapidly spread throughout the system. This leads to the need for investigating "what if" scenarios. These scenarios can be implemented through exploration experiments and techniques.

Exploration experiments and techniques are not intended to validate the system as a whole as much as they validate some of the design decisions made within the MDDS, such as what variables to include in the design vector or the structure of the objective function.

These techniques are important for comprehending the effects of design variables, the shape of a design space, the decisions that should be made while choosing alternatives and the associated consequences. This allows for simultaneous consideration of many dimensions of the problem as well as the management of the design process.

Figure 7.1:

Exploration should be carried out before and after Search and Optimization



A key difficulty in the optimization process is usually the large number of design parameters involved. Many algorithms cannot handle problems of more than 100 variables, and in particular if there is no good and feasible point known to begin with. Such limitations have led to a growing interest in applying design space exploration techniques to limit the size of the design vectors involved.

Furthermore, solutions found may be sensitive to perturbations of the design variables or constraints which might render those solutions as less adequate or even infeasible. Sensitivity experiments should be carried out to investigate the effects of changes in input data on the output results (figure 7.1).

7.2. Pre-Search

Working with design problems with a large number of design variables (parameters) is a difficult task for any optimization algorithm, especially if a good feasible starting point does not exist.

Several pre-optimization exploration processes can, however, be applied to develop an overview of the design space or a region of that space around a specific design point.

Initial points must be analyzed so that an initial design point for optimization can be chosen. In addition, a screening procedure can be implemented to identify critical parameters. These include parameters that have the largest effect on the objective and constraint functions. These parameters define a subset of the original design vector which, if reduced enough, can make optimization more successful (Koch et al., 2002).

In this section pre-optimization exploration processes will be presented. These will include parametric studies and One-Factor-At-A-Time (OFAT) analysis, Design of Experiments (DOE), as well as Latin Hypercubes and Orthogonal Array sampling.

7.2.1. Parameter Studies

Many design studies still rely on sequential parametric studies in which one or two (sometimes three) design variables are changed to examine the effect on the design.

Parametric studies involve analyzing one variable at a time to study the effects of assumptions about particular data. Carpet plots can be used to show the effect of these variables on each of the system constraints or objectives. Carpet plot studies analyze two

independent variables by varying the two items over ranges and evaluating the resultant behavior of another parameter.

The number of parameters *n* in this type of study is limited by the dimensionality that can be perceived graphically and by the 3n growth rate in a number of cases that must be examined using this type of grid searching technique. Still, parametric studies provide visibility into the effects of the parameters that are studied and can yield insight into problems that is not available through more complex multi-dimensional optimization results (Kroo, 1997a).

After specifying each level (value) of each factor (variable), a parametric study can be performed by changing one factor at a time while keeping all other factors at a base level (table 7.1). Each factor is considered at every level and the best result for each factor is selected. The best design is then chosen by extrapolating each factor's behavior.

Table 7.1:

In a parametric study one factor is changed at a time while keeping all other factors at a base level

4 factors, 3 levels each:
$1 + n(I - 1) =$
$1 + 4(3 - 1) = 9\ expts$

Expt No.	Factor			
	A	B	C	D
1	A1	B1	C1	D1
2	A2	B1	C1	D1
3	A3	B1	C1	D1
4	A1	B2	C1	D1
5	A1	B3	C1	D1
6	A1	B1	C2	D1
7	A1	B1	C3	D1
8	A1	B1	C1	D2
9	A1	B1	C1	D3

Using a parametric study, the chances of evaluating the "best design" as part of the study are very low, since interactions between factors are not considered. When the need for a more methodical approach to parameter tuning is acknowledged, designers may attempt a One-Factor-At-A-Time (OFAT) analysis.

Similar to a regular parametric study, OFAT involves tuning a single factor when all others are fixed. However, if the output is improved then the new level is kept for that factor and then moving to the following factor and repeating this process with each factor one at a time (Ridge, 2007).

With OFAT the "best design" is expected to be a member of the

matrix experiment. Unlike with a regular parametric study, with OFAT some interactions between parameters are captured, although the result depends on the order of the factors (de Weck and Willcox, 2005).

7.2.2. Design of Experiments

Design of experiments (DOE) is a statistical technique proposed by Sir R. A. Fisher in England in the early 1920s. Fisher's main objective was to establish the optimum water, rain, sunshine, fertilizer, and soil conditions needed to generate the finest crop. Fisher was able to identify all combinations (*treatments*) of the factors included in experimental study using DOE techniques. These combinations were created using a matrix which allowed each factor an identical number of test conditions. By introducing the DOE technique, Fisher devised the first method to analyze the effect of more than one factor at a time (Roy, 2001).

Figure 7.2:

Multiple combinations of factors and levels are used to analyze the design space

The National Institute of Standards and Technology defines Design Of Experiments (DOE) as:

. . . a systematic, rigorous approach to engineering problem-solving that applies principles and techniques at the data collection stage so as to ensure the generation of valid, defensible, and supportable engineering conclusions. In addition, all of this is carried out under the constraint of a minimal expenditure of engineering runs, time, and money (NIST, 2006).

DOE is a collection of statistical techniques that provide a systematic and efficient way to sample and analyze the design space through the analysis of multiple factors (figure 7.2). The systematic approach is drawn from the methodologies and experiment designs used by DOE. By creating a matrix of runs and using a range of algorithms, the effects on numerous responses can be recognized. The DOE principles of data gathering ensure that only adequate data is collected, improving the efficiency and cost of the experiment (Ridge, 2007).

Some of the capabilities of DOE discussed by (Ridge, 2007) include:

Quantify multiple variables simultaneously: the effects of multiple factors on one or more responses can be studied and examined. This can help identify appropriate factor ranges as well as identify achievable responses and objectives.

Identify variable interactions: the combined effect of factors on a response can be identified.

Identify high impact variables: key drivers among potential factors and their relative importance can be ranked.

Predictive capability within design space: performance of solutions at new points in the design space may be predicted.

These capabilities make DOE an essential approach for any research dealing with large and expensive experiments.

DOE techniques help with the study of many factors simultaneously in an economic fashion. Factor levels are varied so as to maximize the information extracted from the resulting simulations. By studying the effects of individual factors on the results, the best factor combination can be determined (Roy, 2001).

DOE can be used in support of traditional optimization procedures (Koch et al., 2002). DOE techniques and similar strategies are often used before setting up a formal optimization problem (de Weck and Willcox, 2005). Using information obtained from a well-defined DOE study can help optimization models to seek out the best design among many alternatives.

7.2.2.1. Factors, Levels and Responses

A factor is an independent variable chosen from the design vector. The various values at which the factor can be set are known as its levels.

Factors can be divided into either primary or secondary factors. Primary factors, also known as design factors, are those factors that are studied because their effects on the responses are of interest. Secondary factors, also known as held-constant factors, are those factors that are held at a constant value throughout all experiments because they are not of interest in the current study (Ridge, 2007).

The response variable is the output of a certain experiment and represents a measure of the variables of interest.

7.2.2.2. Treatments

A treatment is a particular combination of factor levels. The specific treatments will depend on the experiment design and on the factor's variation range. There are various possible experiment designs. The design will depend on several aspects including the research question, stage of research and the resources available (Ridge, 2007).

Experiments can be represented in a matrix. Within this experiments matrix each row corresponds to one experiment and each column corresponds to one factor (table 7.2). Each experiment corresponds to a different treatment of factor levels that provides an observation (de Weck and Willcox, 2005).

Table 7.2:

Experiments can be represented in a matrix where each row corresponds to one experiment and each column corresponds to one factor

Expt No.	Factor A	Factor B	Observation
1	A1	B1	$ή_1$
2	A1	B2	$ή_2$
3	A2	B1	$ή_3$
4	A2	B2	$ή_4$

7.2.2.3. Effects

Once the experiments have been completed, the results can be used to calculate effects. An effect of a factor is the change in the response due to a change in one or more factors as the level of the factor is changed. There are two types of effects: main and interaction effects.

The main effect of a factor is an averaged individual measure of the effects of factors. It is a measure of the change in the response variable to changes in the level of the factor averaged across all levels of all the other factors.

An Interaction effect is the effect that occurs when the effect of a factor depends on the level of another factor and the combined change in both factors produces an effect greater than or less than that of the sum of their expected effects. These are also called higher-order effects that depend on the number of effects involved.

For example a second-order effect is due to two factors, a third-order to three and so on (Ridge, 2007).

Two other important concepts relating to effects are confounding and aliasing. It is essential to stress the difference between confounding and aliasing. Confounding occurs when it is impossible to separate the effects of two or more effects due to bad experimental planning and implementation, particularly to poor control of factors. Aliasing, on the other hand, is an inability to distinguish several effects due to the nature of the experiment design rather than poor execution (Ridge, 2007).

Several DOE techniques exist. A comprehensive review of many techniques and their use in engineering design is provided by Simpson et al. (1997). In this Chapter two methods will be presented, the full factorial design method and the fractional factorial design method.

7.2.2.4. Full Factorial

The term "factorial" may not have been used before 1935, when Fisher used it in his book *The Design of Experiments* (Fisher, 1975). A full factorial design consists of a crossing of all levels of all factors. It measures the response of every possible treatment combinations of factors and factor levels.

As with any statistical experiment, the experimental runs in a factorial experiment need to be randomized to lower the influence of bias on the experimental results.

Full factorial design provides greater efficiency in the use of available experimental resources and the knowledge learned in comparison to the same number of experimental runs in a less structured context such as OFAT (Czitrom, 1999).

Because the factor levels are all crossed with one another, a full factorial design provides information on the effects of each factor on the response variable which can be analyzed for every main effect and every interaction effect.

Furthermore, the results of a full factorial design are more inclusive over a wider range of conditions due to the combining of factor levels in one experiment.

If a simple factorial experiment contains two levels for each of two factors then it is a 2^2 factorial experiment, because it considers two

levels (the base) for each of two factors (the power), producing $2^2 = 4$ factorial points. The effects of three factors with two levels each can be evaluated in eight experimental treatments that represent the corners of a cube.

$$\#\,levels^{\#factors}$$

Full factorial design is an extremely powerful but expensive method. As the number of factors grows, the number of treatments also rapidly, grows and at some point it overwhelms the experimental resources and becomes infeasible due to high cost. For example, a full factorial design experiment with 10 factors at two levels each will require an expensive $2^{10} = 1024$ treatments.

The full factorial experiment is the ideal design for many design problems, but the size of design spaces limits its applicability. A more efficient design is required if the number of treatments in a full factorial design is too high to be logistically feasible. In this case, a fractional factorial design may be used, in which some of the possible treatments are omitted.

7.2.2.5. Fractional Factorial Design

As mentioned previously, due to the combinatorial explosion and the increase in expense of factorial designs with the increase in the design factors we cannot usually perform a full factorial experiment Instead a subset (fraction) of the possible treatments is carefully considered in a manner that can balance experimental cost with design space coverage.

The subset is selected to utilize the sparsity-of-effects principle. This states that a system or process is likely to be most influenced by some main effects and low-order interactions and less influenced by higher-order interactions (Ridge, 2007).

Fractional designs are expressed using the notation l^{n-k}, where l is the number of levels of each factor studied, n is the number of factors studied, and k describes the size of the fraction of the full factorial used where $1/(l^k)$ represents the fraction of the full factorial design.

For example, for an experiment with five factors and two levels for each factor and choosing k to be two, the fractional factorial design is 2^{5-2} which is 1/4 of a full factorial design. So this experiment requires only eight runs rather than the 32 runs that would be required for the full factorial experiment.

The fractional factorial design can assist in providing information about the most important features of the problem studied while using a fraction of the effort and resources of a full factorial design.

However, there is a price to pay for the fractional factorial's reduction in number of experimental treatments. Some effects will be aliased and therefore indistinguishable from one another. If an alias group seems statistically significant more treatments can be added to separate these aliased effects. This sequential experimentation represents one of the advantages of the fractional factorial.

Depending on the number of factors, and accordingly the design space size, a range of fractional factorials can be implemented from a full factorial. Initially, it may be useful to look at a large number of factors superficially rather than a small number of factors in detail (de Weck and Willcox, 2005).

The methodology to generate fractional factorial designs for more than two levels is very hard and could even be infeasible. Other methods, such as response surface methodology (discussed previously in the analysis section), are more efficient in determining the relationships between the response and factors at multiple levels.

Table 7.3:

In Fractional designs levels are specified for each factor and outputs are evaluated at every combination of values.

2 factors, 3 levels each:
$I^n = 3^2 = 9\ expts$

4 factors, 3 levels each:
$I^n = 3^4 = 81\ expts$

Expt No.	Factor A	Factor B
1	A1	B1
2	A1	B2
3	A1	B3
4	A2	B1
5	A2	B2
6	A2	B3
7	A3	B1
8	A3	B2
9	A3	B3

7.2.3. Latin Hypercubes

The Latin hypercube is a generalization of a Latin square with a larger number of dimensions. A Latin square is a square grid containing sample positions with only one sample in each row and each column.

Latin hypercube sampling was developed in the statistics community and was first described by McKay et al. (1979). It gained interest in

engineering design in 1989 after Sacks et al. (1989) computer experiments.

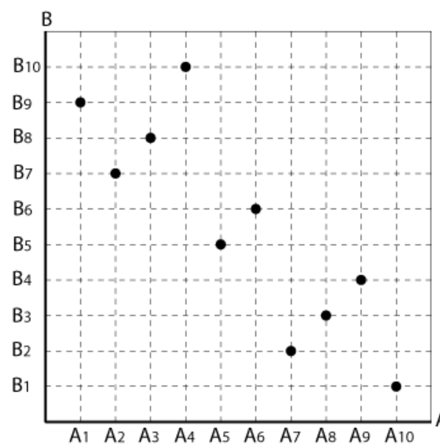
The statistical method of Latin hypercube sampling was developed to generate a distribution of reasonable collections of parameter values from a multidimensional distribution.

A Latin hypercube is a matrix of *M* rows and *N* columns where *M* is the number of levels being examined and *N* is the number of the factors. Each of the *N* columns contains the levels 1, 2, 3..., *M*, randomly permuted. The randomly permuted *M* levels of *N* columns are matched to form the *M* Latin hypercube. When sampling a function of *N* factors, the range of each factor is divided into *M* equally probable intervals. *M* sample points are then placed to satisfy the Latin hypercube requirements. This forces the number of divisions, *M*, to be equal for each factor. Each level of a factor is used only once (de Weck and Willcox, 2005).

Latin hypercube sampling offers flexible sample sizes while ensuring stratified sampling. This sampling scheme does not require more samples for more dimensions (variables); this independence is one of the main advantages of this sampling scheme. Another advantage is that random samples can be taken one at a time, remembering which samples were taken so far (figure 7.3).

Figure 7.3:

An example of a Latin hypercube sampling



The maximum number of combinations for a Latin hypercube of *M* levels and *N* factors (i.e., dimensions) can be computed with the following formula:

$$\prod_{n=0}^{N}(M-n)^{N-1}$$

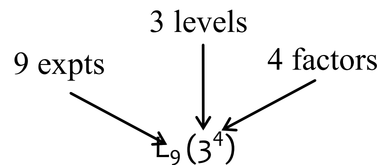
Latin hypercube sampling is particularly well suited for computer experiments, since design points are spread throughout the design space, and more levels are generally taken for each factor than with the other designs (Koch et al., 2002). The approach, however, can produce poor coverage and the results are not repeatable (de Weck and Willcox, 2005).

7.2.4. Orthogonal Arrays

Orthogonal array sampling requires that the entire sample space be sampled evenly. Orthogonal arrays specify levels for each factor. Arrays are used to choose a subset of the full factorial experiment that maintains orthogonality between factors.

Table 7.4:

In the balancing property, for any pair of columns, all combinations of factor levels occur an equal number of times.



Expt No.	Factor A	B	C	D
1	A1	B1	C1	D1
2	A1	B2	C2	D2
3	A1	B3	C3	D3
4	A2	B1	C2	D3
5	A2	B2	C3	D1
6	A2	B3	C1	D2
7	A3	B1	C3	D2
8	A3	B2	C1	D3
9	A3	B3	C2	D1

Although orthogonal array sampling does not capture all interactions, it is still considered an efficient sampling and the experiment is balanced. For any pair of columns, all combinations of factor levels occur and they occur an equal number of times. This is the balancing property (Table 7.4). In general, the balancing property is sufficient for orthogonality (de Weck and Willcox, 2005).

Although more efficient than Latin hypercube sampling, the orthogonal array sampling approach is more difficult to execute since all random samples must be generated at the same time.

7.3. Post-Search

7.3.1. Sensitivity Analysis

Exploration techniques used after search and optimization are mainly sensitivity analysis processes. The importance of sensitivity analysis stems from the fact that all the mathematical models used in the MDDS are approximations to the actual artifact and system (De Neufville, 1990). Some data in the mathematical model are inherently uncertain.

Input is generally affected by many sources of uncertainty including errors in data, lack of information and limited understanding of the design problem. This uncertainty affects our trust in the output of our models. Therefore the examination of the effect of input data on the output results is important.

Sensitivity analysis is primarily concerned with how the specific response of a chosen solution changes due to the modification of design problem formulation. Sensitivity analysis tries to identify what source of uncertainty has a greater effect on the final solutions.

The problem formulation of sensitivity analysis is similar to DOE discussed earlier. In DOE we investigate the effect of some 'treatment' on the performance. In sensitivity analysis, on the other hand, we study the effect of varying the inputs of a mathematical model on the solution.

Sensitivity analysis is key to understanding which design variables, constraints and parameters are important drivers for the optimum solutions. Using sensitivity analysis techniques we can verify the effects of changing the design variables, parameters and constraints on the “optimal” solutions selected. There are several sensitivity analysis techniques such as simple derivates, sampling and screening, Monte Carlo filtering, and variance based approaches among others (Helton et al., 2006).

8. MDDS

8.1 What is the MDDS ?

In the previous chapters I discussed different concepts that represent the stages needed to construct what I call the Multi-Disciplinary Design System (MDDS). In this chapter I will attempt to tie these concepts together into a coherent framework.

The MDDS represents a design process rather than a specific design tool. The core of this process involves creating integrated computational systems. There are five steps to generating an MDDS: decomposition, formulation, modeling, integration, and exploration. These steps are not carried out in a sequential manner, but rather in a continuous back and forth between the different steps as the design progresses and evolves (figure 8.1).

As discussed earlier, design can be seen as both an object and a process. This has strong implications for how we decompose an artifact in the attempt to build the MDDS, since we have to decompose both the artifact object as well as the design process that was used to produce it. In formulation, many tools have been suggested for the task of structuring and formulating the information produced from the decomposition stage into a coherent MDDS architecture. Methods for activity modeling that include synthesis, analysis, evaluation and optimization were discussed in the modeling chapter. Later in the integration chapter different integration software technologies were suggested to connect the different activity modules. Finally, in the exploration chapter different tools and techniques were discussed for the task of design space exploration.

In this chapter I will expand on these ideas and will propose a framework that demonstrates the processes and relationships involved in building the MDDS. I will then discuss how this system will evolve as the design of the artifact evolves. Concepts for handling complexity will be discussed including multi-variable, multi-module, multi-levels, and multi-resolution processes.

After discussing the MDDS evolution, I will then discuss the expected

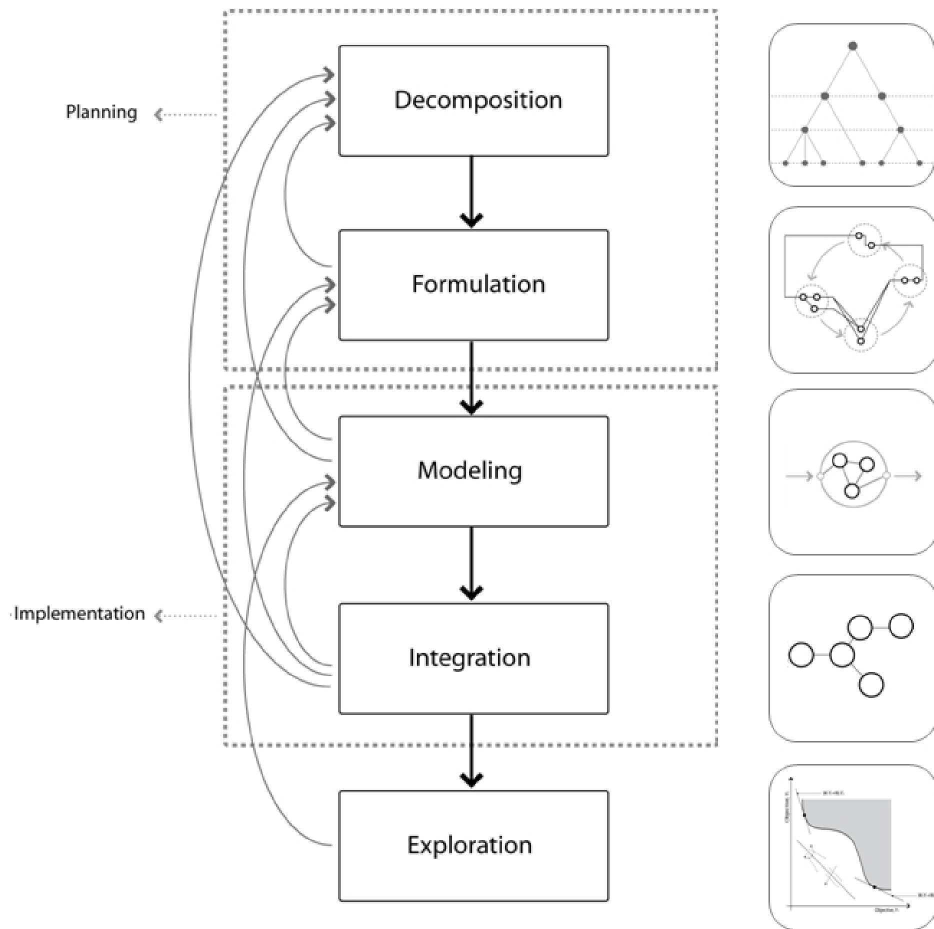
behavior of the system. This will include the system's emergent qualities and its multidisciplinary and performance-driven behaviors. Finally, I will discuss the affects of using such a system on structuring the design team and the implications for developing new design tools and environments.

8.2 MDDS Framework

The design team must identify the design concept that can best perform the design requirements. Specialists from many fields work with the system architect to ensure that the solution considers all specialty requirements that the system architect may not have been fully aware of. If there is more than one concept involved, the alternative concepts are usually swapped among each other in order to come up with a preferable solution.

Figure 8.1:

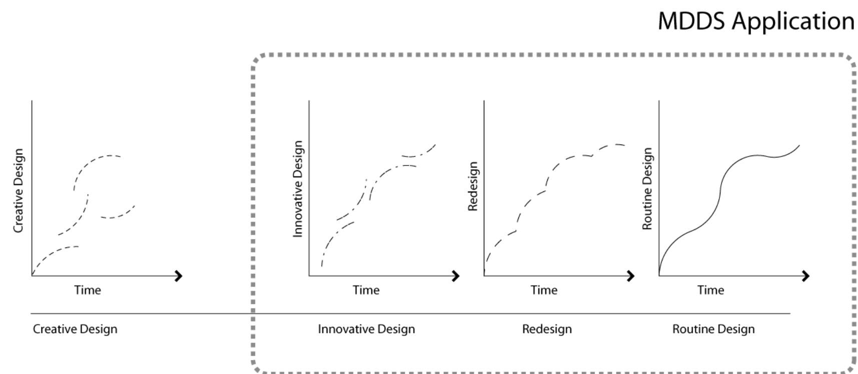
MDDS Framework includes five phases: decomposition, formulation, modeling, integration, and exploration



As stated earlier, the MDDS is applied after a design concept has been developed. As mentioned earlier, Duvvuru et al. (1989) divided the design process into four basic categories: creative design, innovative design, redesign, and routine design. I believe the MDDS can be used for routine design, redesign, and even innovative design. However, creative design, which involves the initiation to create the main design concept, has to be carried out by the human design team (figure 8.2). This is typically the most difficult stage of the whole design process, but as the design space narrows down incrementally downstream, later stages become more defined and more suitable for the MDDS implementation.

Figure 8.2:

MDDS application in relation to Duvvuru design categories



While developing the concept, the design team can define and suggest the design vector variables and even the main objective function to be used in the MDDS, although there will be a need to further refine them in later stages. The resulting design is then integrated in the MDDS and is used as a point-of-departure reference to provide a rough estimate for the design of the artifact or system in order to start the MDDS building process.

The first stage in building the MDDS is *Decomposition*. Here the artifact or system design concept is broken down into, on one hand, the different components and aspects that make up its physical object, and, on the other, the developmental levels and design activities that can be used to construct the design process. A comprehensive list of disciplines and information required from each discipline for design and development should be established. The design team should also define the initial design vector variables, in addition to establishing preliminary objective functions if they have not been established yet. In this stage, the starting point for the process of problem formulation and modeling is setup.

Then, the *Formulation* stage defines how various modules will be interconnected. In order to arrive at a reasonable system architecture, there must be an iterative cycle or loop between decomposition and formulation. The input and output parameters should also be explored for each of the involved activity modules. The formulation of this system architecture can lead to important enhancements concerning design turnaround times in addition to significant minimization of unnecessary additional performed tasks (Atherton, 2002).

MDDS offers a framework where the modeling of complex design problems can be achieved by aggregating sub-problems. The design system is typically modeled in terms of modules which are interacting objects that represent individually specific design activities. The *Modeling* stage is where these mathematical models are developed. These modules can contain engineering models in addition to data or software applications. Furthermore, both the design vector variables and the objective function are better defined in this stage but can still be modified further according to investigations made in the design exploration stage.

In the *Integration* stage, the modules are integrated such that the necessary design information is passed between them. With modules being able to represent various parts of the problem simultaneously, the integrated MDDS is realized as a computational design tool capable of producing design solutions.

Exploration is the fifth and final stage of the MDDS framework and occurs after the MDDS is fully developed and verified. Automated multivariable parametric studies and trade studies can be conducted to evaluate the design vector and the design objectives. Furthermore, the sensitivity of the solutions to the design constraints can be studied. Design vector parameters and the objective function are varied in this stage in order to explore the design space.

8.2.1 Decomposition

One of the basic assumptions of the proposed framework involves the fundamental idea of decomposition. This decomposition process takes place at the front end of the MDDS construction development. Initially, through a top down approach the design concept will be decomposed iteratively by each discipline involved in the design. The decomposition is carried from a high-level diagram ending in manageable smaller sub-problems. These sub-problems facilitate, in addition to the problem solution, a better understanding of the

design problem domain.

Decomposition strategies have been discussed earlier in the decomposition chapter. Two modes of decomposition were presented, namely the object and process decompositions. Both modes are essential for building the MDDS (figure 8.3).

Process decomposition is composed of both development and activity decompositions and represents the main elements needed later in the formulation stage. Development decomposition informs the formulation stage about the proposed hierarchy and multilevel structure of the MDDS. Activity decomposition on the other hand is essential in identifying the design activity modules within every level.

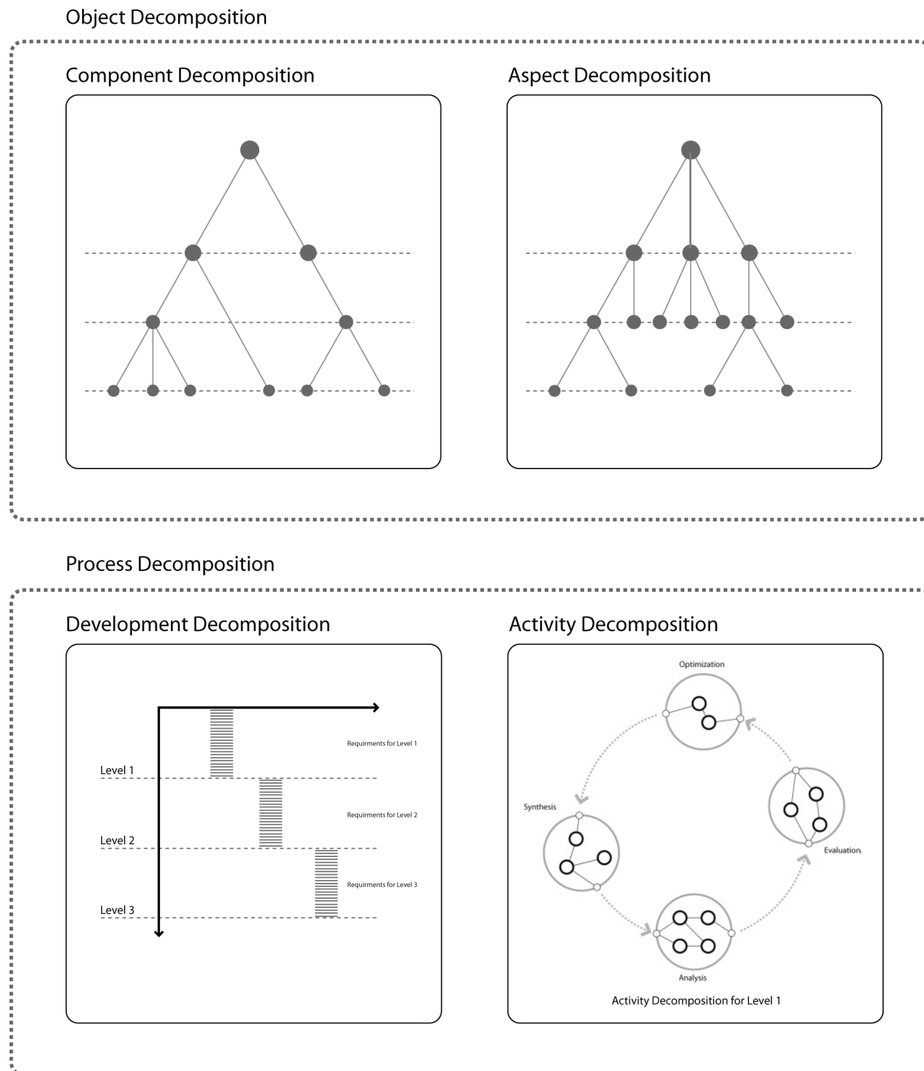


Figure 8.3:

Object decomposition includes both component and aspect decompositions while process decomposition includes both development and activity decompositions

Furthermore, object decomposition is needed in the modeling stage. As discussed earlier object decomposition includes component and aspect decompositions. Component decomposition divides the system or artifact according to the physical system elements. Component decomposition is essential to the synthesis modeling. Aspect decomposition, on the other hand, treats the problem according to the system physics and therefore is critical for the analysis models.

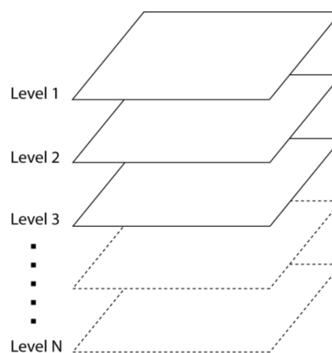
8.2.2 Formulation

In this stage the architecture of the MDDS is created. To represent the MDDS we use different levels of aggregation of complex interacting elements. Similar to decomposition, while formulating the MDDS we have to make choices on the level of abstraction needed. High abstractions do not usually require domain knowledge, and are therefore used to summarize, generalize, and compare. Low abstractions require domain knowledge, and thus provide valid details where differences are explicable. In our context formulation is primarily based on the idea that MDDS encompass a number of levels and design activities.

A system that comprises design activities with high complexity cannot be easily or efficiently managed as a monolithic entity, and so it has to be broken down into development levels. Therefore, the MDDS is broken into hierarchical levels in order to manage design complexities, where each lower level becomes more detailed and refined as the design progresses (figure 8.4).

Figure 8.4:

MDDS is broken into hierarchical levels

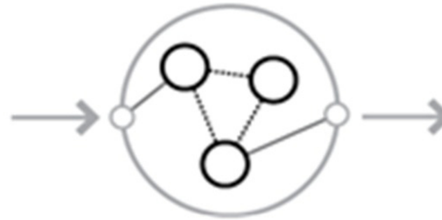


MDDS is made of modules, where each module represents a design activity. Similar activity modules can be interconnected to create assemblies. MDDS comprises a group of modules and cycles which represent a system at different levels of abstraction and also possibly at multiple development stages. The system architecture identifies which modules will be part of the system and provides descriptions

of their roles. The MDDS as a whole can be seen as a set of interrelated modules that collectively can produce design solutions (figure8.5).

Figure 8.5:

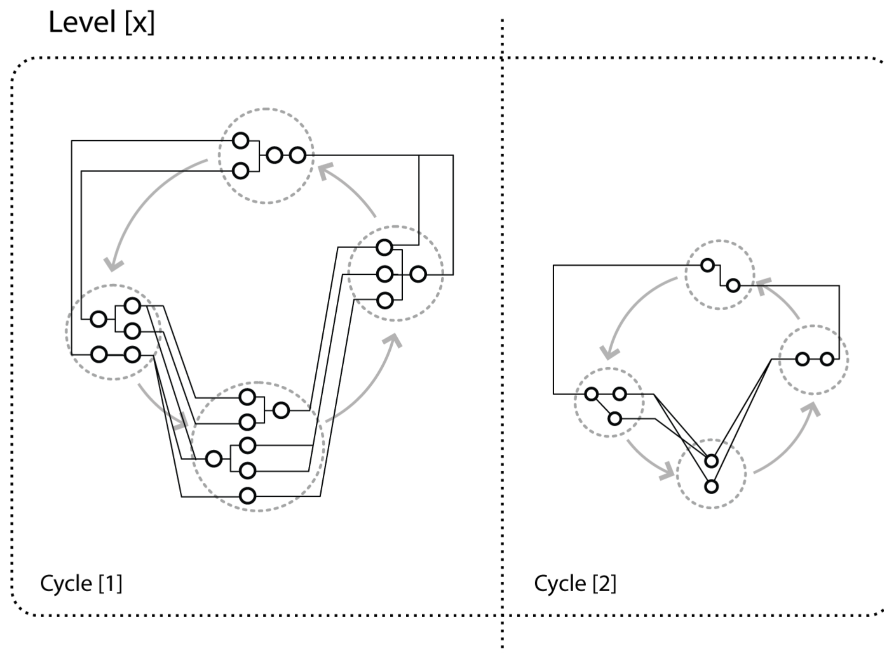
MDDS comprises a group of modules



Iteration is a basic concept in all design processes. Researchers have discussed different approaches for managing these iterations (Smith and Eppinger, 1997). I refer to iterations within the context of MDDS as design cycles, were each cycle includes all four design activities mentioned earlier, namely synthesis, analysis, evaluation, and optimization. Through a bottom-up approach, the design activity modules are connected into a design cycle that represents a data flow network. Each design cycle resides in a design level within the MDDS (figure 8.6).

Figure 8.6:

Each design cycle resides in a design level within the MDDS



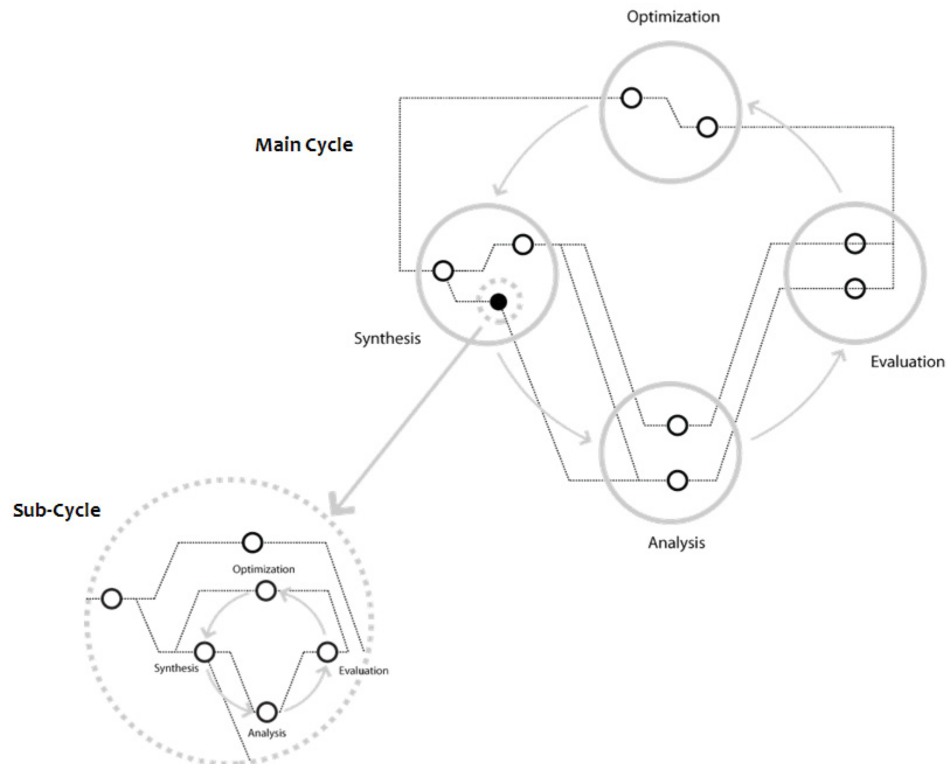
Given the descriptions above, it is clear that the MDDS includes both hierarchical and non-hierarchical structures. Within its hierarchical structure, it is possible to define discrete tree-like interaction

patterns which offer well-guided navigation within the process. This hierarchical layout enables multilevel problem formulation. The MDDS levels are of such a structure. These hierarchies can also be layered hierarchies where horizontal relations could be established within a single design level and between two or more design cycles.

The non-hierarchical structures define relations between the different elements within an MDDS level. These elements include modules, assemblies and design cycles. A *data flow network* is created between the different elements with links that represent the interactions between them. These links allow the flow of information between the different modules.

Figure 8.7:

A design cycle can include sub-cycles



The MDDS also evolves and grows over time, either vertically by adding more levels or horizontally by adding more modules and cycles (figure 8.8). This will be discussed further in the Evolution section of this chapter.

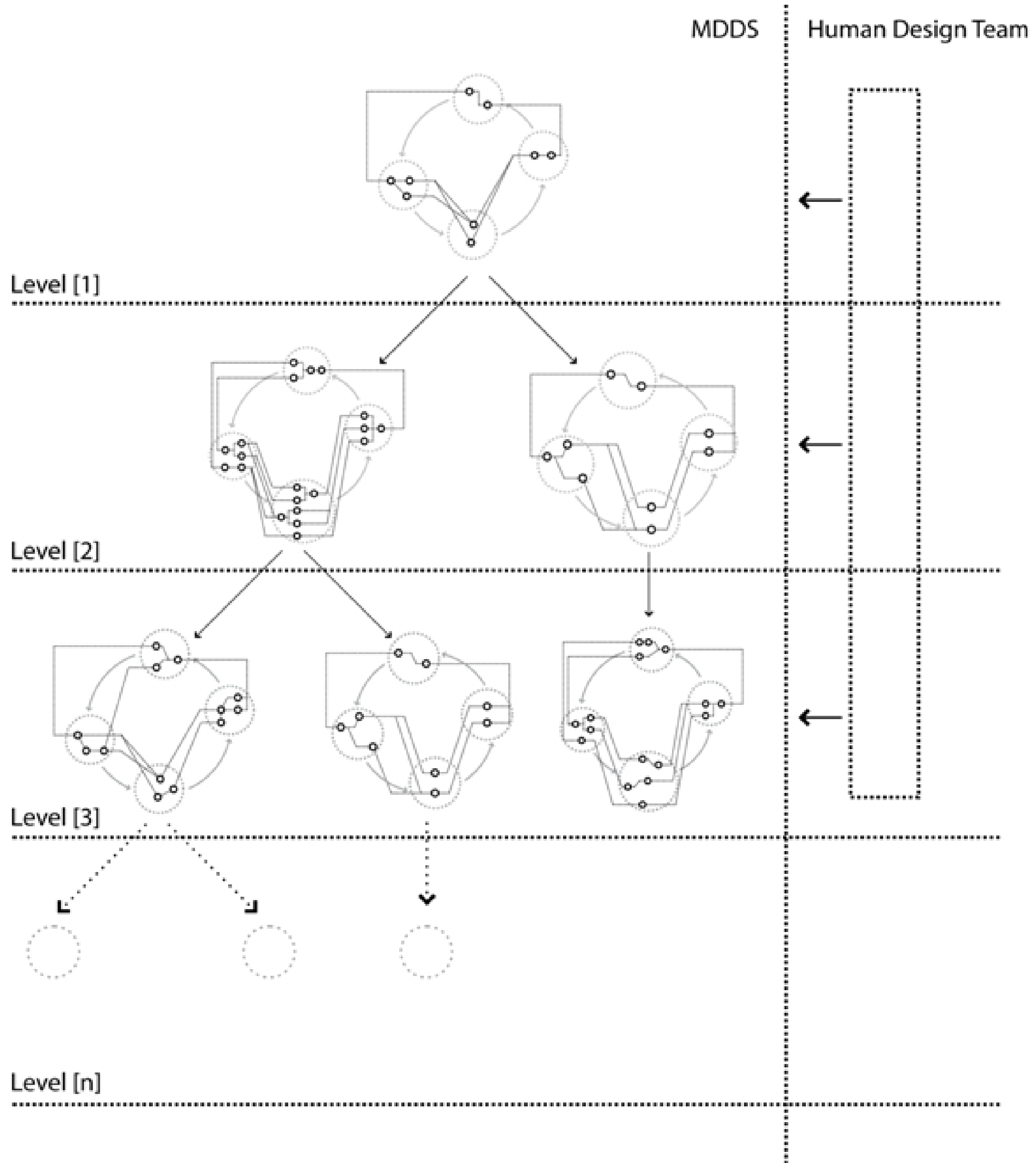


Figure 8.8:

The MDDS evolves and grows over time, either vertically by adding more levels or horizontally by adding more modules and cycles

The MDDS architectures developed in this stage should be fed back to the decomposition process in order to fine-tune the lower levels of functional analysis according to the evolving higher-level solutions. This is a process which should be done with careful pacing.

The tools and notations discussed in the formulation chapter can be useful in formulating the order of activities and interactions in the MDDS. The DSM for example, could be used to refine the interaction between modules and minimize iterations as well as determining

crucial activities that influence process lead-time and cost. Formulation notations that include network notations, such as Data Flow Diagrams or IDEF0, or even formulation modeling languages, such as UML and SysML, can be of great use in designing the MDDS architecture and defining its hierarchical levels, cycles, assemblies and module interactions.

In summary, formulation works on promoting the interaction among the system architects, design specialists and other design team members. It occurs prior to modeling and programming in order to avoid major reprogramming later on. Formulation also enables the visualization of data and control flow. This is very useful for the system architect, as it affords him a lot of time and effort in examining the MDDS model.

8.2.3 Modeling

As discussed earlier in the modeling chapter, models are abstract descriptions of the real world that provide approximate representations of more complex functions of physical systems (Papalambros, 2000). Many design problems require using a group of complementary models, instead of one single model, which together aim at modeling and describing the whole design problem. The modeling process that encompasses many issues in large problems requires specialized knowledge in many disciplines (Pahng et al., 1997). No single designer can excel in all these disciplines, hence there is a need for different people who have the suitable principal competencies to model and solve different aspects of the design problem (Eppinger, et al. 1994).

Within the MDDS we are concerned with mathematical models. These are models that can be implemented in a computer environment. We aim at building a mathematical model for each activity module. These include the *synthesis, analysis, evaluation,* and *optimization* activities (figure 8.9). Modules for data storage and constrains can also be included. These are modules that store the parameters and constants and the design system constraints. Extra modules for data flow control could also be implemented.

Each module has a boundary that cuts across its links to the environment defining that module's input and output. Each module acts like a black box transforming data from one form to another. The behavior of each module contributes not only to the design aspect and discipline it is modeled after, but to the design system as a whole.

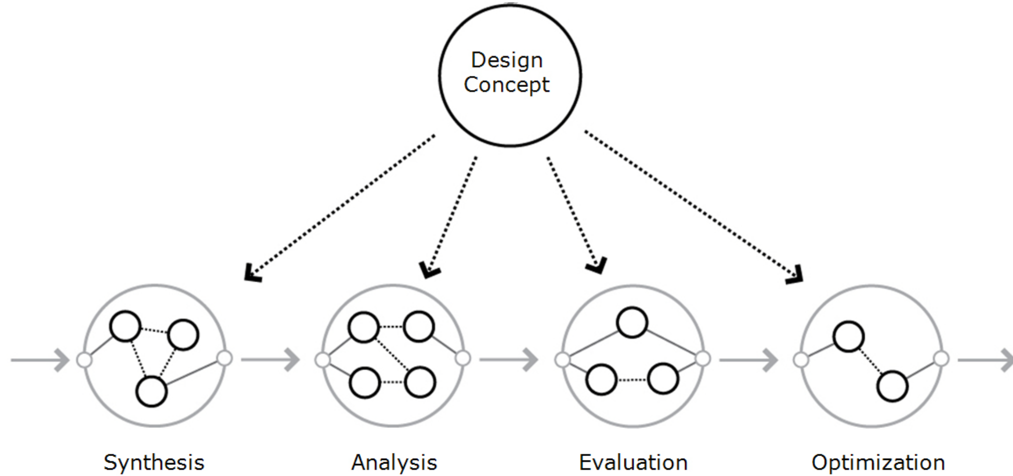


Figure 8.9:

A design cycle that regenerates a design concept should include synthesis, analysis, evaluation, and optimization activities

Domain knowledge of each discipline involved in the design informs the synthesis modules to create meaningful designs and representations. The outcome of the synthesis modules is analyzed by the different discipline analysis modules to predict the properties of a particular solution. The evaluation modules then handle the multi-objective nature of the design. The optimization modules search the design space and automate the synthesis, analysis and evaluation in search of new solutions. The process continues until the optimization has converged and a family of acceptable solutions is found.

Modeling can take place through one of two basic approaches: programming the model in a programming language such as C or C++; or constructing it in simulation software such as CAD, FEA, or CFD. Using programming languages provides better program control and a low purchase cost. Simulation software however minimizes programming time and thus lowers project cost. After constructing the model, it is validated to make sure the original assumptions were acceptable.

The scope of the model is primarily based on the fidelity degree needed at a certain MDDS level. This is an essential issue in modeling activities and will be discussed further in the MDDS Evolution section.

8.2.3.1 Synthesis

The synthesis mathematical model defines the system configurations to be modeled. These models are influenced by the component decomposition completed in earlier stages. The design concept is decomposed into a set of synthesis models by extracting design

intentions and formulating a collection of design parameters, rules or algorithms. This collection provides for a representation of the design language which in turn defines a design space. This mode of representation provides for a formalism that can be used within a computational environment to breed new designs.

The design vector or variables within it are the input to this type of module. As discussed previously in the modeling chapter, the number and type of variables included in the design vector depends on the algorithms and structure of the synthesis model. Synthesis modules can offer precise feedback for the MDDS on the influence of parameter variations within the design vector on geometric data.

Synthesis modules output data to analysis modules. This data includes design attributes such as dimensions, areas, volumes, locations, vectors, and mass properties. The need for integrating synthesis and analysis modules affects to a great extent the modeling requirements for both design activities.

Synthesis models should provide for a generative mechanism. This could be done through the different techniques discussed in the modeling chapter, such as parametric and algorithmic models. Parametric models provide for a description of the artifact through parameters and relationships that allow for variation. Algorithmic models provide a description of the artifact through a set of rules and algorithms. Some good examples of algorithmic models are formal Grammars. These include grammars like Shape Grammars, Graph Grammars, Lindenmayer Systems, and Cellular Automata.

The representation of generative synthesis models should encode design knowledge. The relationship between form and performance should be embedded within the representation formalism. This provides restrictions on permitted designs and ensures that the rules discard designs that do not comply with constraints. However, since synthesis models do not include performance feedback loops, it is difficult for such models to direct the generation and navigation of the design space of multi-performance design problems.

Furthermore, the geometry resulting from the synthesis process must be robust enough to cope with the intense variations that take place in later trade studies. After all these modules come into place, the utility of each module is evaluated by conducting various verification cases and design studies (Atherton, 2002).

8.2.3.2 Analysis

An analysis model infers from a design solution characteristics that are relevant to a particular discipline. A design problem usually combines different disciplines with each discipline developing one or more analysis models.

The outcome produced by a synthesis module is the input to the analysis module. These may range from simple parameters and data such as areas or volumes, to full CAD models for use in numerical analysis like FEM and CFD. The outputs of the analysis module are performance and behavior measures that will eventually be used within an evaluation module to assess the effectiveness of a system configuration.

In the modeling chapter several analysis models were discussed. These models range in their amount of required information input and their degree of accuracy output. Analytical models are mainly low-order (low-fidelity) models that are fairly fast but with low accuracy. On the other hand, numerical models like finite element analysis (FEA) and computational fluid dynamics (CFD) are high-order (high-fidelity) models which have higher accuracy but result in long durations which has a compound effect when such a model is run several times in a design exploration and multidisciplinary optimization process. Many low-processing approximation concepts have been utilized to generate surrogate behavior models to replace expensive and detailed analysis and simulation software when testing numerous scenarios with various input parameters (Koch et al., 2002; Bletzinger and Lähr, 2006).

In choosing a model the designer must select the best compromise between the demand for simplification and the necessity to clearly identify, describe and rate the targeted physical mechanism. A trade-off will have to be made between fidelity and analysis time.

8.2.3.3 Evaluation

The need for the evaluation of results arises while observing systems in multidisciplinary contexts. Evaluation modules are in essence decision-making tools. The input to an evaluation module is the output of several analysis modules. Evaluation therefore refers to the overall result of a design analysis, which encompasses multiple analysis computations.

The output of the evaluation module depends on the strategy used in the evaluation and whether the evaluation is done before or after optimization. An evaluation is usually performed by means of an

objective function which consists of a figure of merit describing the quality of a design solution. The formulation of the objective function is vital to the outcome of the design space search. A solution is expressed in an n-dimensional design space. "n" relies directly on the number of design objectives. Results from the evaluation module usually yield a dimensionless quantity known as the quality for each solution.

In order to make a decision about rationally choosing one of the alternatives, a criterion is required which assesses all alternatives and ranks them in a certain way. The criterion, which is called the objective of the model, cannot be unique, as its choice is usually affected by several factors. These factors include the design application, timing, point of view, and designer judgment and may change with time. (Papalambros and Wilde, 2000).

In single objective optimization, the search direction can be well defined and a single solution, if it exists, could be found. However, in the real world, design problems are usually too complex and ill-defined and have several possibly contradicting objectives. This implies that there is no single optimal solution but rather a whole set of possible solutions of equivalent quality. In this set, each objective is optimized with the understanding that if any further optimization is attempted, the other objectives could be affected as a consequence. Therefore, decisions need to be taken in the presence of trade-offs between conflicting objectives.

Addressing multiple objective problems may require techniques that are different from standard single objective optimization methods. This evaluation of multiple objectives is articulated based on the decision-maker's preferences either before or after the search.

When the preference is expressed beforehand, the designer decides how to aggregate different conflicting objectives into a single objective function before the actual search is performed. A commonly adopted approach is scalarization which consists of combining several objectives into one scalar cost function. There are different scalarization methods, such as the weighted-sum approach and the utility function method among others.

When search is performed before decision-making, the search is performed with multiple objectives at the same time. The solution space becomes partially ordered with a set of optimal trade-offs between the conflicting objectives. This set is called the Pareto optimal set.

8.2.3.4 Optimization

The final step in the design cycle involves optimizing the design to investigate the performance benefit increase. Many configurations can basically meet similar design goals. Thus an optimization problem can be put forward in order to search for an optimum configuration. Each configuration has its individual group of design variables and functions. This implies that a design can be changed to provide various alternatives (Papalambros and Wilde, 2000).

The goal of optimization studies in this context involves studying how a design performs and how this performance can be influenced in order to choose the most desirable alternative or alternatives (Bletzinger and Lähr, 2006).

Optimization modules are design space search machines. Searching the design space entails finding the best solution(s) within a domain of feasible solutions. The choice of an appropriate search algorithm depends on several factors, including the design synthesis model, the nature of the analysis models, the number of design variables, the existence of constraints, and the linearity of either the design variables or constraints.

The input to the optimization module is an objective function that depends on a number of continuous or discrete values. The optimization module seeks to minimize or maximize an objective function by varying the values of those variables within an allowed domain. The outputs of the optimization module are new values for the design vector variables.

As discussed earlier optimization algorithms could be divided into discrete numerical optimization techniques or heuristic algorithms. Some numerical optimization techniques that handle constraints include the simplex method, sequential quadratic programming, and the exterior and interior penalty methods among others. Discrete numerical optimization techniques that handle unconstrained problems are generally gradient-based algorithms. These include Newton's method, steepest descent, and conjugate gradient among others. Within the interconnected and highly nonlinear nature of multidisciplinary design problems, it cannot be supposed that a given solution is globally optimal merely because it may be locally optimal (Atherton, 2002). Conventional gradient-based methods may not be suitable for this purpose, since they locate the optimum solution according to the point in the function space at which they started. On the other hand, heuristic algorithms are generally non-gradient methods, like evolutionary algorithms, simulated annealing, and tabu

search, can escape local optima. However, no existing optimization technique is guaranteed to find the global optimum of a nonlinear, non-convex problem.

Gradient-based methods find local optima with high reliability but might not escape a local optimum. Heuristic algorithms might find a good solution, but its optimality cannot be guaranteed since they often tend to find a different design each time they are run. In addition, they do not converge to a solution in the same effective manner as gradient-based methods do.

Furthermore, no single optimization technique is applicable in general to all types of engineering design problems. Studies in the field of nonlinear constrained problems, which are common in complex engineering design problems, have demonstrated that no single optimization technique performs best for the majority of design problems.

For a given design problem, a combination of techniques often performs better than single techniques. Using the two dissimilar methods in a complementary way creates a 'hybrid' optimization strategy that can address the problem efficiently. This strategy would ideally promote relative strengths of both methods and restrain their weaknesses in order to provide maximum analytical benefits. A heuristic technique, for example, can be applied to a problem with a high degree of nonlinearity and multiple predicted local optima to globally identify within the design space regions where best solutions may lie. Starting from the solution or solutions obtained in this exploratory search, a numerical technique can then be applied to search locally for the best solution in this specified region of interest, or also to fine-tune it. The most effective way however to solve a given problem will always be dependent on the specifics and details of that unique problem (Koch et al., 2002).

There still remain some issues when novice users apply optimization techniques in complex design problems. These include choice of the starting point, the number of system analyses required for optimization, uncertainties in problem formulation and design parameters, and their effects on the optimization (Koch et al., 2002). Other questions and challenges relative to optimization exist at both the system and discipline level. How to handle the problem of multilevel optimization, how the optimum solution is established, what it is sensitive to, how robust is it, and how to determine if it is really the optimum solution will be discussed in the MDDS evolution section.

8.2.4 Integration

Integration takes place at the tail end of the MDDS development. Integration mainly aims at facilitating the coupling of activity modules and simulation programs regardless of discipline, programming language or format (Koch et al., 2002).

As discussed earlier the idea of a single super software is not really compelling practically when it comes to building simulation tools to cover a wide range of disciplines. This software cannot simply be tailored to address in detail any single domain within its range of applicability. If only one set of tools exists, one analysis process, or one design philosophy, then there would be very little space for genuine creativity and innovation.

The benefits of smaller interacting components and modules and the component-assembly approach in the software industry have been recognized in many applications and have resulted in recent focus on component-ware (Kroo, 1997). Software modules enable their design specialists to exchange and discuss design information, alterations in design tasks and design decisions with other specialists.

Selected modules that were modeled and created by design specialists are then assembled and integrated in the MDDS (figure 8.10). Interfaces describe the group of services that a module can provide (Pahng et al., 1997). They demonstrate detailed descriptions of how different modules interact together. This includes how the modules fit together, connect, and communicate. If these interfaces are compatible, modules can consequently interact with each other. The interface analysis process is useful as it aids in refining the architecture along the lines of minimized cross-organizational interfaces (Grady, 1994).

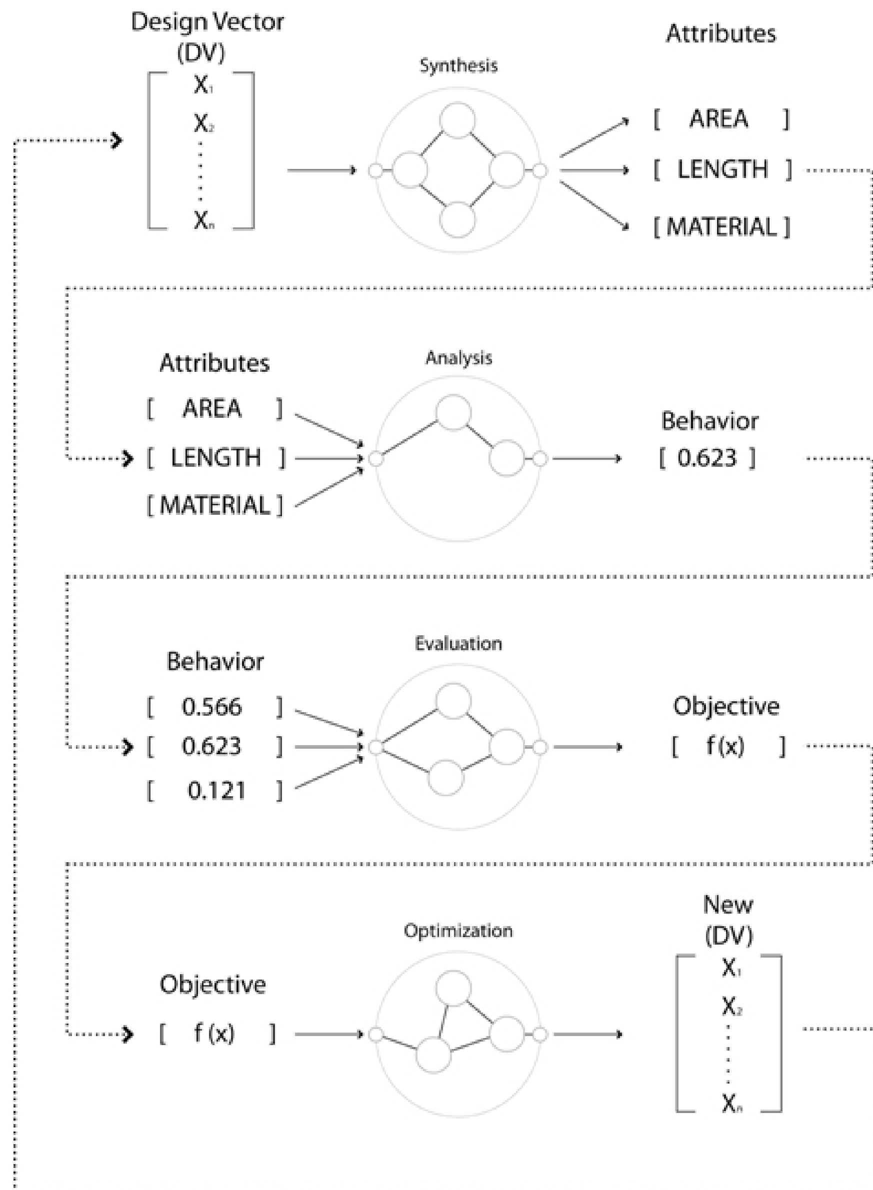
The integration between the different modules can be carried out using one of the integration technologies discussed earlier in the integration chapter, such as middleware, web services or a combination of both. The system architect decides on the data that will be shared from one module or tool to the next so as to assemble an efficient MDDS. This data should pass between modules in an automatic fashion as soon as it is all linked together. The design teams can then focus on the design problem independently.

Managing dataflow from one module to the next, which has always been extremely time-consuming in the past, would be overcome by providing execution scheduling functionality and easing module communication. Design information that a certain module desires to receive represents module interests. These interests could trigger

the action of receiving the well-suited design information as soon as it is generated by any other module. Implementing automation within the MDDS would surely minimize the time required to run design iterations.

Figure 8.10:

The different design activity modules are integrated in the MDDS



Furthermore, the MDDS should be built on the idea that all system variables should be accessible and monitored from a central location or framework, but the modules that distribute these variables should be free to run on any platform independent of the controlling framework (Atherton, 2002). As a whole, the organization of design teams is considered as a distributed design environment where both design specialists and software modules of different design teams are geographically dispersed.

There are certain risks that the MDDS may not actually work as formerly planned. Testing the system involves running the simulations and reviewing the model validity. With increased experience, system architects can predict more of these risky and negative interactions until the integration task becomes much easier.

8.2.5 Exploration

After building and integrating the MDDS, it would be useful to carry out a few experiments that could help explore the design space. Exploration experiments and techniques are not intended as a validation of the system as a whole as much as they are a validation of some of the design decisions made within the MDDS, such as what variables to include in the design vector, the systems constraints or the structure of the objective function.

A key difficulty in the optimization process is the large number of design parameters involved. Many algorithms cannot handle problems of more than 100 variables, and in particular if there is no good, feasible point known to begin with. Furthermore, optimization studies typically need multiple computer iterations which may be expensive or time consuming, especially in the case of large, complex systems. These limitations have led to a growing interest in design space exploration techniques (Koch et al., 2002).

Design space exploration can delve into "what-if" scenarios and assess trade-off situations. This makes it an essential tool for analyzing the effects of design variables and the shape of design spaces, providing a better understanding of the decisions that are made in design selection and the corresponding consequences.

Exploration techniques such as DOE can be used to provide an overview of the design space or a local region of the design space around an initial design. DOE concepts define a systematic and efficient means by which a design space is analyzed, basic design variable screening is provided, design variable effect is evaluated, and important design variable interactions are identified (Koch et al., 2002). The parameters that have the largest effect on the objective

and constraint functions identify a subset of the original design variables set. Optimization can become more feasible if this set is reduced in number. A new and feasible or enhanced initial point for optimization can sometimes be chosen using the initial points analyzed from the DOE study. Approximations of the original analysis or simulation programs can be generated using the full set of DOE points.

Sensitivity analyses can also be utilized to identify what model factors have key influence on performance measures. This implies consequently modeling those factors carefully. Sensitivity analysis is primarily concerned with how the specific response of a system changes due to the modification of some other specific input parameters (Bletzinger and Lähr, 2006). This includes exploring the design space for the solution sensitivity with respect to input parameters at specified design points. The degree of dependence of the result, for example, from the input parameters or possible extrema regarding those parameters can be assessed. Sensitivity relies in its computation method on the partial derivation of the quality regarding the input parameter (Bletzinger and Lähr, 2006).

Traditionally, conducting a multidisciplinary trade study is characterized by being a time consuming process which is largely dominated by the reformatting, transforming, and translating of data between design disciplines and analysis modules (Atherton, 2002). However, the MDDS approach can offer the design team the flexibility in addressing dissimilar and large trade-spaces by allowing the quick interchange of individual modules, leading to the easy testing of the effect of these modules on the design solutions.

8.3 System Evolution

8.3.1 Complexity

One of the basic issues in engineering and designing systems is the issue of complexity. There are several definitions of complex systems. A complex system is defined by Crawley et al. (2004) as a system that comprises components and interconnections, interactions, or interdependencies, all of which are hard to describe, understand, predict, manage, design, or change.

In design, managing complexity represents a huge challenge. Nature however, offers the most compelling examples in relation to complex system design, since they are the outcome of an evolutionary design process that encompasses ever-changing complexity.

Similarly, the nature of the MDDS process is one that involves evolution. This notion of an evolving system yields an MDDS that is continuously dependent on and responsive to the uncertainties of design progress.

The MDDS design development decomposition recognizes the evolution and the hierarchies inherent in the design process. The MDDS design should be viewed as an incrementally changing process that grows from the top to bottom as a combination of multiple quasi-interdependent levels. Therefore, the MDDS resulting system model can be described as an evolutionary model.

Designers move from simple and generic designs into more complex and detailed ones throughout the design process. Early on in the process, the exact structure of design objects is not clearly defined (Rosenman and Simoff, 2001). With project progress, the design description must evolve and change, as well as the constraints and synthesis and analysis models. Practically, the level of description of a specific design should be directly proportional to the amount of information available at a specific project stage. A design could not be described at the fabrication level when the project is still at an early stage, as too much information impedes the project's progress.

That is primarily why design development is divided into conceptual, preliminary, and detailed design (Kroo, 1997a). Design description complexity and mathematical model sophistication increase as more detail is added, moving from simple representations to more detailed descriptions.

Although the design development process appears to be sequential

with steps following each other, the reality is that certain knowledge can be gained or some circumstances can change as the process moves forward, thus questioning decisions early on in the process. The selection of properties or design vectors may change based on specific knowledge acquired along the analytical model development (McManus et al. 2004). Most MDO efforts do not take the evolution of design complexity in consideration. They are only limited to the problem of minimizing a specified function according to an assigned group of design parameters (Kroo, 1997a).

The MDDS can be described as encompassing a dynamic architecture and structure which is affected by several factors. These include the number of modules and cycles needed at a particular design level and phase. Additional factors also include the number and type of variables in a specific design vector, as well as the required degree of fidelity. Therefore, in addition to being multi-disciplinary, the MDDS can be characterized as comprising multi-levels, multi-modules, multi-variables and multi-resolutions. These characteristics are described below (figure 8.11).

8.3.1.1 Multi-Level

Hierarchical levels can be identified in the system definition. Each system is decomposed into subsystems. These can be further decomposed with the different subsystems being linked together. The analysis of each system occurs at a specific level of complexity that is compatible with the interests of the individual who studies the system.

Both the artifact design and the design process can be viewed in terms of hierarchical decompositions, where they are decomposed into multi-levels. Therefore, MDDS should also be considered as a multi-level hierarchical system.

Effective planning is required, however, where there is an evolution from one level of maturity to the other. Hence, at each level of the MDDS, design problem decomposition and formulation should take uncertainty of lower levels into consideration.

Solution coordination is an important factor in achieving a solution to the full design problem through multiple solutions for the decomposed multi-levels. Through the formulation of design problems at different levels of the decomposed problem and the transfer of information across these levels, the MDDS goals can be reached.

8.3.1.2 Multi-Module

Modules are distinct abstractions that have simple interfaces. The abstraction hides the complexity of the modules, while the interface indicates how that module interacts with the larger system (Baldwin and Clark, 2000).

This notion of abstraction is very much related to the concept of information hiding, which was first introduced by David Parnas in software engineering (Parnas, 1972), and is applicable to any complex system. Parnas argued that if the details of a specific block of code were deliberately hidden from other blocks, alterations to the block could take place without changing the rest of the system. Designers should then divide the design parameters into two main categories: visible information and hidden information. This will indicate which parameters interact outside their module, in addition to how potential interactions will be managed across modules.

Therefore, an MDDS, at a certain level, will comprise several modules. These modules are combined in assemblies and cycles. There could be several assemblies in one cycle as well as several cycles within one level.

Given the multi-module and multi-level characteristics of the MDDS, the state of the system should be observed both horizontally and vertically.

8.3.1.3 Multi-Variable

The design vector and its set of variables evolve and change between the different levels of the MDDS. In this evolution, some variables might continue to evolve by continuing to vary in subsequent levels; others might be locked and thus removed from the design vector, while other new variables might be added to the design vector. The number of variables within the design vector is known as the number of degrees of freedom.

As discussed earlier, specific degrees of freedom should be enabled within a design cycle for the purpose of experimentation. However, redesign of the design vector(s) is required in successive levels. This redesign process must be strongly built on the experience acquired from working on different MDDS architectures.

These observations are significant for a suitable MDDS cycle definition which will constitute the core for building its different mathematical models. Depending on the number of cycles per level, the MDDS can be represented as a single design vector for a single

cycle on a level or as a group of design vectors in different cycles that should be synchronized and managed at a global system level. This becomes critical to modeling decisions as the system increases in size.

8.3.1.4 Multi-Resolution

In early conceptual design stages, MDDS can be used to synthesize many alternatives, and pertinent analysis can be conducted. In later phases, however, more detail is required to perform elaborate synthesis and analysis. These are conducted using higher-fidelity modeling and tools. A simple system such as one that incorporates beam representations of structures can be easily modeled. But, when it is substituted by a plate or finite element model, the number of design degrees of freedom and system dimensionality increase remarkably. With this evolution, higher-fidelity analysis is often required (McManus et al. 2004).

For the evolving MDDS, modules with different resolutions and granularity levels are needed. By altering modules or exchanging existing disciplinary synthesis and analysis modules for more suitable fidelity levels, existing MDDS level models can be evolved to lower successive levels.

Furthermore, the nature of the design problem itself can change with design progress. In emergent situations, initial design vectors, parameters and models may become irrelevant. In order to move forward with identifying solutions and exploring design spaces, relevant models have to be identified and instantiated. This involves dealing with more and more complex design parameters and results, which increase computation time, making the enhancement of the fidelity of disciplinary analyses a difficult task (McManus et al. 2004).

Multi-resolution can be implemented in two directions: vertically and horizontally. As discussed above, vertical multi-resolution takes place between the different levels of the MDDS. On the other hand, horizontal multi-resolution can occur within one design cycle. For example, two modules could model the same aspect with one module running at a higher-fidelity level, and therefore taking a longer time to run, while the lower-fidelity module runs faster but does not provide accurate answers. In this case, the system architect could use the faster low fidelity module within the optimization, and at different intervals of the optimization verify its results using the higher fidelity module (Similar concepts were discussed earlier in the modeling chapter).

However, these multi-resolution representations will have various

modeling needs that can intensify the design challenge. The primary concern of multi-resolution modeling is resolving representational discrepancies that evolve among modules (Davis and Bigelow, 2002). Having different modules working at different levels or within design cycles implies the need to preserve consistency at each abstraction level. Reynolds et al. (1997) discuss the challenges in this process. Design strategies that take these potential discrepancies into consideration are necessary for designing these cross-resolution models.

8.3.1.5 Decoupling

Decoupling within the MDDS takes place when the interactions between parts of the system disappear. This happens when the various interconnected disciplines, aspects and analyses are decomposed into subgroups which do not require the output of another group as their input. The system structure is thus simplified and can benefit from parallelism.

Horizontally: Modules and Cycles

Modularity, as mentioned earlier, is a specific design structure where parameters and tasks are interdependent within modules and independent across them. Modules in a larger system work together as units but are structurally independent of one another. This implies that a module's internal structural elements are strongly linked among themselves but weakly linked, with gradations of modularity to elements in other modules (Baldwin and Clark, 2000). This notion of independence and interdependence of modules must be identified for any design. The system architecture must allow for both the independence of structure and the integration of function.

Coupling or dependency within this context is defined as the degree to which each module relies on each one of the other modules in a given system (Kroo, 1997a). Low coupling, or "loose" or "weak", denotes a relationship where one module interacts with another one through a stable interface without any concern about the internal implementation of the other module. Thus a change in one module does not require a change in the implementation of the other one.

High coupling, or “tight” or “strong”, occurs if one module changes or relies on the internal implementation of another module, such as accessing local data from it, and so the dependent module will change according to manipulations in the way the other module produces data. This is also known as content coupling (Kroo, 1997a).

The term decoupling is thus used in a single design cycle to identify

the segregation of modules that should not be dependent on each other. Doing so usually minimizes the degree and risk of failure in any one part of a system if another part is altered.

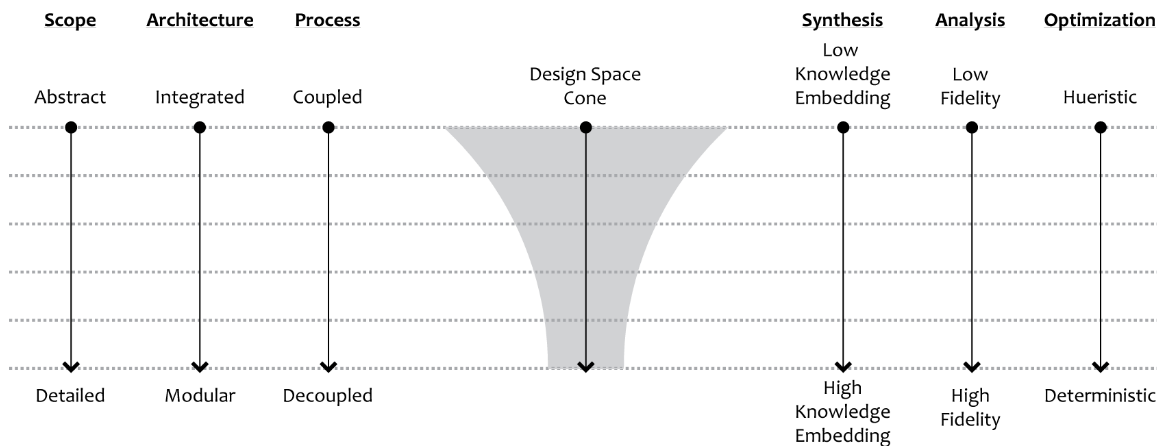
Vertically: Levels

Here we describe the decoupling that happens between successive levels within an MDDS as the system evolves. As the design of an artifact progresses, physical parts and components and their associated functions and aspects that are weakly related and have very little influence on other components and aspects can be synthesized and analyzed individually.

A design cycle within a specific level that is intended to generate certain component configurations can therefore evolve in subsequent lower levels of the MDDS into two or more decoupled design cycles. Furthermore, new cycles and new modules may be created as new levels surface in the MDDS. Therefore, the MDDS architecture is expected to be integrated in higher design levels and modular in the lower levels.

Figure 8.11:

MDDS captures design evolution



8.3.2 Adaptability

Holland (1992) thoroughly discusses in his book *Adaptation in Natural and Artificial Systems* the concept of adaptation or the adaptive process, which involves progressively changing a certain structure. He states that a set of structural modifiers or operators is generated through carefully observing successive structural modifications. Consequently, the observed modification sequences are generated through the repeated action of these operators.

These operators represent actions in complex adaptive systems that modify existing structures into new structures in well-defined ways.

Operators in this sense are similar to verbs in a language or functions in mathematics. They identify a group of paths by which the system can change, evolve and become more complex using their powers of conversion (Baldwin and Clark, 2000). Holland highlights the fact that a system that involves the combination of operators that act on structures at every stage is a system that experiences adaptation.

Within the MDDS, modularity is a concept that demonstrates adaptation. Modularity has been be very useful in many domains involved in complex system design.

Within the MDDS many design cycle, options can be generated using the modular mix-and-match flexibility leading to a final design cycle that suits its current needs. In this dynamic process, and as new modules are tested and incorporated into larger design cycles, the MDDS as a whole will start to change and evolve.

There are several features and actions that can take place among modules. Four main actions can be done in an MDDS: a module design can be *substituted* by another; a new module can be *added* to the system; a module can be *deleted* from the system; and a module can be *reused* in another model.

Modules that share common services can be swapped to investigate diverse solution alternatives in a problem model (Pahng et al., 1997). This potential of swapping or substitution constitutes the core of economic competition, which can be only reasonable if two different modules can serve the same ends, but not equally well (Baldwin and Clark, 2000). If knowing in advance which module will be better is not possible, then both modules can be generated and tested against each other. The essence of substitution in this context lies in the notion that the better design will win this competition.

In MDDS, a user can select at the beginning a set of modules to meet his or her needs. In this context, MDDS is said to be *configurable*. It can also be *reconfigurable* according to changing needs, where the user can augment (or add) modules to the system to give it some new kind of functionality, or exclude (or subtract) modules that are no longer needed (Baldwin and Clark, 2000). MDDS is also considered reconfigurable through the substitutions mentioned earlier which represent the upgrading of existing modules. Modules can also be potentially reusable in other problems.

Reusing well-structured and generic modules significantly minimizes the time required to build models for new design problems. After the construction of building blocks within the design process, these blocks can be saved to a library to facilitate its reuse and sharing

among other projects. They can also be reused as templates for generating similar activity modules (Koch et al., 2002).

By supporting interchangeability, modularity enables designers to control how changes in processes or requirements affect the designed product. The flexibility by which they can meet these changing processes enables them to delay design decisions until more information becomes available (Gershenson et al., 2003).

In this context, an existing MDDS at a specific level can evolve to a lower level through changing or replacing existing modules for those with better-suited fidelity levels. In addition modules can be added or removed from the design cycles in that level.

8.3.3 Optimality

Within the MDDS a set of optimization tuning parameters can be established for each design cycle and level. These could include parameters like the maximum number of iterations or convergence criteria among others. The optimization can be carried out in a multi-step decomposed optimization plan that integrates various levels of the MDDS. Using a successive filtering of solutions, certain solutions are moved from one level to the next to be optimized further. This is a sequential optimization technique between the different levels which is not expected to necessarily lead to an optimum solution.

However, the question of optimality within this context is debatable since it depends on many factors including the initial design concept, what is included in the design vector, the analysis modules implemented, the objective functions and constraints applied among many other factors. Given an initial concept the goal of optimization within MDDS is to guide the evolution of the design towards solutions with higher performance and not necessarily to an optimum solution.

One of the difficulties associated with optimization in MDDS is the high level of uncertainty involved between the levels since, in many cases, the lower levels are not yet known. However, after the full design has evolved and further optimization of the full system is sought, several multi-level optimization techniques can be implemented.

Three of these methods have been studied in detail, including: concurrent subspace optimization (CSSO), collaborative optimization, (CO) and Analytical target cascading (ATC).

CSSO depends on partitioning the design problem into various

subspaces pertaining to the specific disciplines. Each of these subspaces attempts to minimize a global objective, while at the same time sharing responsibility for the satisfaction of the system constraints. This process is managed differently by system level algorithms according to the implementation.

CO also deals with simultaneous subproblem optimization. Disciplinary teams, however, are responsible for satisfying local constraints in the process of attempting to meet the target values assigned by system coordinators. These shared target values are tweaked by the system in order to reduce some objectives while enabling the subspaces to meet those targets. The CO method is appealing in many domains due to the simplification it offers in analysis integration and communication. It also allows the domain specific selection of optimization algorithms. It is particularly appropriate for analyses already coupled with optimization. There are still some limitations to existing CO applications including slow system-level convergence and sensitivity to subspace feasibility tolerances (Kroo, 1997b). In general, however, this method has great potential for large problems with low dimensionality interdisciplinary coupling.

Analytical target cascading (ATC) is another technique developed for hierarchical multilevel system optimization. This methodology, unlike the case with MDOs, basically addresses hierarchies that are decomposed by objects or physical subsystems and not aspects or disciplines.

ATC methodology is based primarily on the idea that the performance of a system element can be derived analytically as a function of its decision variables (Choudharyet al., 2005), and therefore performance goals can be embodied as design targets, which can be accomplished through design decisions. Some performance goals can be defined as global design targets, and proposed as part of the initial problem definition. The compatible targets and performance specifications can be derived computationally as functions of design decisions by using analysis and simulation models. Concurrent design can thus be achieved by solving the sub-problems in isolation in more detail (Choudharyet al., 2005).

8.3.4 Time

To gain the maximum benefit from MDDS, design iteration time should be significantly minimized so that many solutions can be achieved in the process as a whole. Usually the design team spends less time executing or specifying information pertaining to design

and analysis and instead spends most of the time in managing that information.

MDDS can have a significant role in minimizing design iteration time through a set of methodologies and technologies. These include mathematical modeling, systems approach, integrated design schemas, automated synthesis, discipline analysis and multidisciplinary optimization, which all lead to enhanced performance in both process and product.

This reduction in iteration time enables design teams to explore the performance of many more alternatives during conceptual design than what is now possible, thus leading to potential improvement in initial cost, performance and overall quality results. Many concepts can be analyzed in parallel and related trade studies can be conducted to investigate the design space.

This resultant efficiency does not however come without a price, particularly when it comes to setup time. A lot of initial investment is required in setup time and process planning. This is compensated for though throughout the life cycle of the design system. Through controlling previous design activity processes and modules during model construction, the scale of this investment can be significantly reduced.

8.4 System Behavior

The MDDS should primarily work on maximizing performance. However, since each discipline involved in the design can have more than one performance attribute or requirement, a balance should exist between these performance attributes, although conflicts are highly expected. These conflicts are not expected only within a single discipline, but also between different disciplines. The notion of emergence through conflict is thus clear, as the attempt to resolve these conflicts usually produces unexpected solutions known as emergent solutions.

8.4.1 Performance Driven Design

A design process that is usually driven by meeting a group of budget, constraints, and functionality criteria often produces an end-product that merely satisfies these minimum criteria. The design team has to work collaboratively in order to accomplish performance that is better than just average. Establishing performance goals early on in the design process augments these efforts and makes it easier to achieve these better results.

Performance based design provides the basis by which design is guided through performance. This provides an all-inclusive methodology to artifact design through embracing a set of performance-based priorities and simulation technologies of analysis. This widespread scope of performance-based design implies crossing many worlds, including the financial, cultural and technical. The MDDS approach introduces a scenario where this idea of performance driving design is clearly identified.

The analysis in this approach will not rely on the artifact's original geometric definition, but the geometry itself will reflect analysis results. This is due to feedback loops from analysis to synthesis using optimization. This definitely represents a more efficient approach than the process where design synthesis takes place first followed by working exhaustively on what the real artifact is required to be (Carty and Davies, 2004).

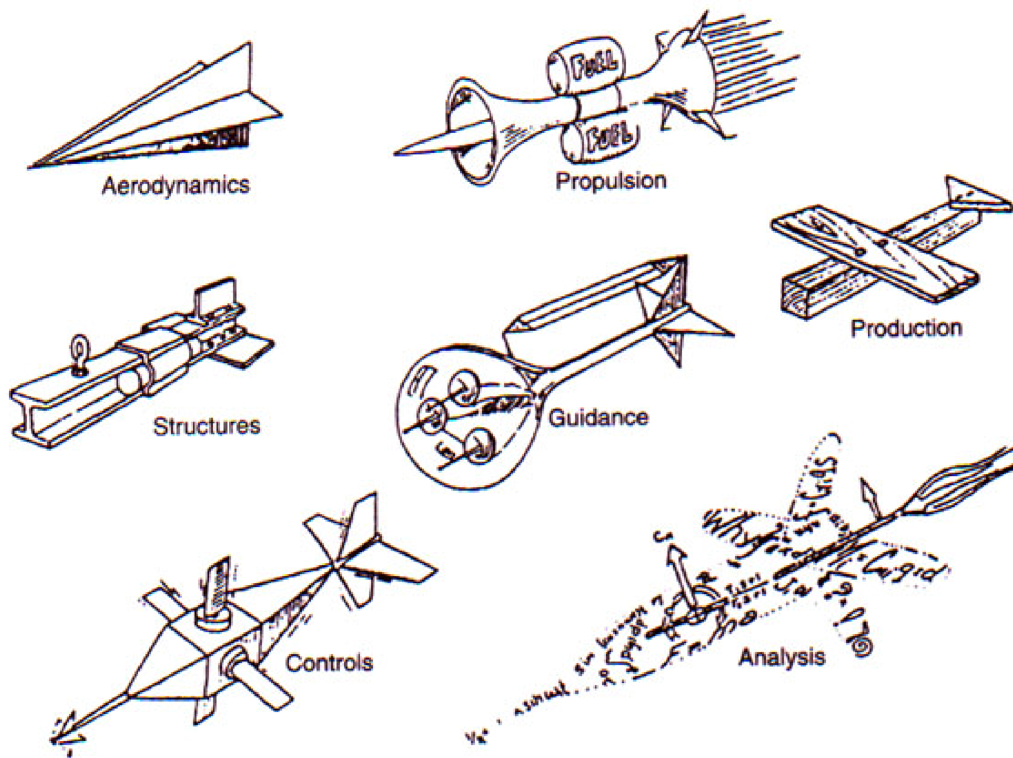
Applying these methods should establish designs with high performance, but should not disregard or overlook the vast diversity in design configurations and physical characteristics. Generated designs with similar performances are likely to have in some cases remarkable differences in their form and configuration. Therefore, the performance-based approach has significance and potential success in not only promoting high performance but also catalyzing the design process, as it demonstrates that similar performances can

be attained through many different ways. Within the MDDS performance goals and search procedures can be set and applied to identify which design features are closest in achieving the desired targets. The actual payoff in this situation arises when this approach is used repeatedly and automatically.

The dilemma remains in specifying which performance criteria we should be optimizing for. It is clear that although various disciplines and design specialists are involved in an artifact design, these specialists mostly aim at optimizing specific aspects of their own discipline that they best understand (Sydenham, 2003). Figure 8.12 reflects such an aspect, as it illustrates how a guided missile is conceived from the perspective of each design specialist individually, and how each optimizes the system aspect that suits their discipline.

Figure 8.12:

A guided missile is conceived from the perspective of each design specialist individually



8.4.2 Collaborative Multidisciplinary Perspective

Designing complex systems is a multidisciplinary process performed by design specialists who could possibly be geographically dispersed. These specialists use a variety of design activity modules and software to achieve a common purpose (Khedro, 1996). Results of one analysis module in the design process often affect results in other analysis modules. At the same time, evaluating a design effectively requires the integration of multiple disciplines (Atherton, 2002).

Design specialists mostly focus on issues that are directly related to their area of technical expertise and responsibilities although they understand that artifacts and systems are groups of components that offer a specific set of capabilities in combination. On the other hand, systems architects must focus continuously on system design globally. Their way of attending to design specialty issues is valid as long as it addresses global performance, developmental risk, cost, or long term system viability. It is therefore the role of the systems architect to orient system development in such a way that guarantees that the appropriate balance is achieved between attention and resources while reaching optimal system behavior.

Usually in the design process, design specialists first satisfy the difficult design requirements and constraints while performing design tasks in order to guarantee correctness of the design. Then they move on to optimizing the design from their viewpoint through satisfying other soft constraints and criteria (Khedro, 1996). These soft constraints can be changed during the design process and do not represent a fundamental factor for achieving a safe and sound design. As long as the design solutions satisfy the hard design constraints regardless of fulfilling all soft constraints, complex system design problems remain under-constrained problems.

Design specialists tend to make and communicate design decisions based on the performance of many design tasks, both in a synchronous and asynchronous fashion. Cases can happen however where the decisions they make conflict with the hard or soft constraints of other specialists due to their limited knowledge of them. In this case, the specialists can relax only their soft constraints in an attempt to resolve those conflicts and arrive at a shared and reasonable agreement about the design. The hard constraints however cannot be relaxed.

Khedro (1996) classifies design conflicts into two main types: critical and non-critical. Critical design conflicts are an outcome of hard

constraint violation. These always have to be resolved, as it is necessary to satisfy hard constraints to arrive at feasible designs. Non-critical design conflicts are an outcome of soft constraint violation, and therefore these constraints have to be relaxed by the design specialist to reach a reasonable agreement.

Therefore, one of the basic functions of building an MDDS is to constitute a state of balance among the different disciplines and design specialists involved in the artifact design. The MDDS therefore implies significant focus on balance, in an attempt to certify that no particular attribute can thrive at the expense of an equally important or even more important attribute, such as performance growing at the expense of reasonable cost.

Addressing a design problem from a multi-discipline perspective allows a more genuine understanding of the system level design trade-space than does a myopic view of individual discipline impacts on the system (Atherton, 2002).

Several tools that address conflicting criteria were discussed in the modeling and exploration chapters. However, regardless of the tools used the outcome cannot be predicted nor expected but is rather emergent.

8.4.3 Emergence

In this section, we examine the concept of emergence which is one of the main concepts for developing and understanding MDDS. Views regarding the concept of emergence are influenced by explorations in the disciplines of evolutionary biology, philosophy of science, cybernetics, systems theory, and artificial life. This is due to the ambiguous role of a concept such as emergence. (McCormack and Dorin, 2001).

Emergence is a broad term comprising hardly related meanings within different disciplines. This makes it harder to clearly define and even understand. Every author has offered his or her own classification of emergence and its various forms, making little room for agreement between individual authors, as well as between disciplines. There is continuous debate about defining emergence in terms of linguistic, epistemic or ontological constructs (McCormack and Dorin, 2001).

The concept of emergence originated in the nineteenth century where it was studied in fields of physical, chemical and biological systems. The common interpretation of *emergence*, which does not belong to any special domain, denotes the revelation, appearance, or

the action of making visible of an event, object or outcome of any process. In design, emergence represents novelty, surprise, or spontaneity as well (McCormack and Dorin, 2001).

An important difference has been pointed out between emergent properties that can be explained in terms of products of lower level interactions, and others that cannot. This implies the concept of the whole being *more* than the sum of its parts. In addition, emergence is only recognized *after* it takes place as it cannot be hypothetically predicted (McCormack and Dorin, 2001).

After assembling the MDDS, it becomes a dynamic and complex whole, interacting as a holistic structured functional unit that searches the design space for satisfactory solutions. The system emergent properties are not detectable through the properties and behaviors of its modules, and can only be enucleated through a holistic approach. The solution found by this system is expected to be superior to the design found by solving and optimizing each discipline sequentially, since it can exploit the interactions between the disciplines. While MDDS is active, emergent or spontaneous patterns can materialize as different forces compete.

One of the very intriguing issues about emergence in the context of MDDS is that it establishes an attractive methodology that addresses Descartes' Dictum: "how can a designer build a device which outperforms the designer's specifications?" (Cariani, 1991). This is due to its strong relation to qualitatively novel structures and behaviors that are not reducible to those hierarchically below them (Channon and Damper, 1998). This leads to creating designs that can that cannot be predicted by an MDDS creators. Therefore the MDDS can be described as having intelligent behaviors.

8.5 MDDS Team and Environment

8.5.1 MDDS Team

The desire to integrate the work of multiple disciplines is rooted in basic principles of design and engineering, as large complex systems comprise components that are of interest for several disciplines and technologies. These systems must satisfy many complex needs that cannot be reconciled with simple solutions. However, humans are faced with two basic challenges pertaining to the production of technology and information. Humans are limited in the information, knowledge, and technology that they can manage and excel in. At the same time, the knowledge generated in these processes surpasses by far human individual limitations (Grady, 1994). Those who make use of all available knowledge resources can economically solve more complex problems than others who can only hold smaller knowledge bases. With the evolution of technology, mankind comes across innovative and complex challenges, and thus promotes more complex combinations of the available technology. To meet those challenges Grady (1994) argues that the general solution to the dilemma between the human capacity and the available knowledge resources lies in the specialization solution.

This solution involves the specialization of individuals in limited disciplines while devising means to bring together the skills and talents of a team of specialists to constitute a different type of specialist, namely the system architect. The truth of the matter is that it would be more efficient and less chaotic if a single human mind could tackle a design problem rather than multiple people trying to work together. There is no doubt, however, that a system created by a well-led team of specialists would have the upper hand when put into comparison with one created by a single individual (Grady, 1994). A team, consisting of design specialists and system architects, therefore would jointly be capable of grasping a larger body of knowledge and experience.

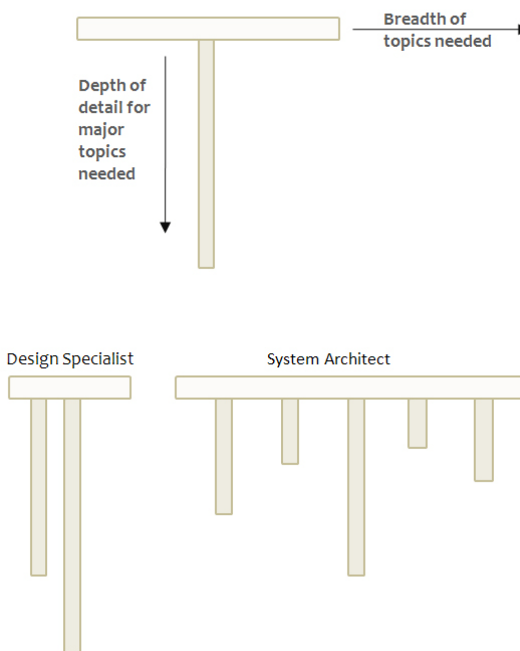
The importance of design specialists here lies in gaining maximum advantage from their expanded experiences. The role of design specialists is to guarantee that their share of the requirements and constraints in the design process are matched by their performance of specific design tasks and suitable design decisions (Khedro, 1996). These tasks include building modules and assemblies that perform design activities. There is a need, however, during the process, to identify how a design specialist is affected by the design decisions of another specialist. In this multidisciplinary environment, specialists can be knowledgeable of their own discipline, but limited in

knowledge about others. They would therefore not know how the system would respond if some parameters in their modules were altered (Bletzinger and Lähr, 2006).

The role of system architects is to perform the task of system architecture design and integration. This involves putting modules together into cycles, making sure that they function together as a system, and investigating possible failures if the system does not functionally operate. System architects should have a broad spectrum of knowledge, as they will be continually asked to assemble multiple types of systems. It is of extreme importance to know how deep that knowledge should be. It cannot obviously be equivalent to the knowledge of the design specialists in their detailed disciplines (figure 8.13). It should be wide enough to take into account several factors, such as risks, technological performance limits, and interfacing requirements, and sufficient to enable performing trade-off analyses among design alternatives (Kossiakoff et al., 2003).

Figure 8.13:

Knowledge domains of systems engineer and design specialist.



The knowledge of system architects should extend from the highest level of the system architecture and its environment down to the lowest level system building blocks and modules. In parallel, the knowledge of the design specialist should extend conversely from the lowest level of modules, upward passing through the full functional level of the module (Kossiakoff et al., 2003). These two

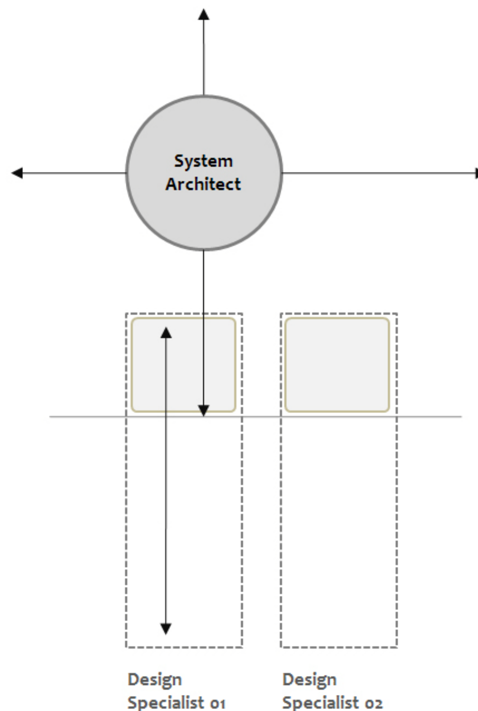
knowledge domains intersect and overlap in this manner such that the system architect and the design specialist define the various technical problems through effective communication. They also discuss and negotiate acceptable solutions that take into account the general capabilities of the artifact and the MDDS design process.

Responsibilities of both system architects and design specialists can be defined by means of the MDDS hierarchical structure (figure 8.14). Modules and sub-cycles denote elements that that lie within the domain of design specialists who can adjust them to a specific application given a group of specifications.

System architects should be able to manage the complexity of formulating the system architecture. The number of levels, the number and type of activity modules and the technical tradeoffs that will influence system capabilities must all be resolved by the system architect. Interface conflicts must also be settled to arrive at a balanced design across the system as a whole. Systems architects should be ready to learn enough about the behavior of modules or sub-cycles that are critical to the operation of the MDDS in order to detect their possible influence on the entire system.

Figure 8.14:

Responsibilities of the system architect and design specialist intersect.



8.5.2 MDDS Environment

As discussed previously, component-based software engineering and the component-assembly approach for design have been evolving and become increasingly appealing in the software industry. Both the design workspace and process are affected by this approach. Instances of modules and components are brought together to produce the MDDS cycles and levels. As the MDDS is essentially a program designed as an assembly of linked components, the environment that can help create the MDDS can be considered a virtual design studio that implements the component-assembly approach.

Therefore, this component-assembly approach identifies the role of the system architect as an assembler who links and puts components together. Different representations are acquired when assemblies are dealt with as components or modules. The view of the design specialist who initially designed the assembly is distinguished from that of the system architect who treats that assembly as a module.
In this environment, creating an MDDS does not require writing extensive code to link or create programs. The MDDS environment should offer tools that manage the interaction of software components. The environment should also preferably work on standard computing infrastructures to allow for cross-platform code integration.

The MDDS design environment should enable generating integrated models by allowing all design participants to embed collaboratively their specific software tools or models into modules. Hence, each module acts as a standalone software component. The services it can offer and the services that must be offered to it are pointed out through its public interface.

In this environment, the modular format in which the integration of models and software applications occurs enables users to select well-suited models based on the required level of fidelity in addition to being tailored to address the type of design problem. This allows designers to focus on the problem as the generated data is linked and passed automatically between analysis modules.

This environment can provide powerful management tools that can be applied to computation-intensive design activities. An internal dataflow management can be assigned for such an environment.

Data flow visualization is also needed in an MDDS developing environment. This would afford the user a reflective view of the

model during the MDDS assembly process. The environment therefore should be augmented with a user-friendly graphic user interface (GUI) that encompasses module data linking and post processing.

The environment should inform the system architect, who acts as the central dataflow manager, of events that affect the dataflow of any iteration. The automated data transfer that results will certainly participate in minimizing the time consumed in evaluating and evolving designs.

The environment should also implement groups of design exploration tools for design space exploration. In addition, solution monitoring, and result visualization and post processing should be sustained.

Other capabilities that could also be implemented and empowered by automated tools include generating graphs, conducting optimization studies, creating reports, or viewing 3D model representations.

In addition, the environment should be able to store modules developed by design team members for use in similar design problems. The module can also be made publicly available to other design participants. In this manner MDDS environments could benefit from business models that are more widespread within subclasses of the technology business sector, such as the direct business model, which is utilized by many companies including Dell. This could also benefit from the increasing popularity of the World Wide Web by publishing models on the Internet (Pahng et al., 1997). Furthermore, the environment should be capable of running distributed models (Atherton, 2002).

Within the MDDS environment the synthesis, analysis, evaluation or optimization modules can be published by experts and organizations as live services available through service marketplaces over the Internet.

Such a marketplace can offer a new paradigm for design development. The competition between different firms will lie not only in their ability to design new products, but also on developing mathematical models and embodying and publishing new design services. In this context, designers and suppliers come closer together as a result of highly responsive modeling and simulation systems (Abrahamson et al., 2000).

9. Experiments

I will now demonstrate the applicability of the *Multi-Disciplinary Design System* through some experiments that I hope will express the strength of the MDDS in generating interesting design solutions. Riccardo Merello collaborated with me on all these experiments while Philipp Geyer collaborated on experiment one level two.

The first experiment will demonstrate the ability of the MDDS in handling complexity through evolution and decoupling horizontally and vertically. The second experiment will demonstrate the adaptability of the MDDS framework.

9.1 Experiment 1 - Level 1

9.1.1 Concept

The design concept of our experiment includes a simple allocation of discrete but interdependent *spatial components* within a rectangular site that is divided into an *n* x *m* grid of *cells.* These *spatial components* are wrapped within a *skin* that defines their interface with the environment. When a configuration of the spatial components is reached, a structural frame is generated. The spatial components are allowed to relocate and deform to satisfy multiple performance and objective requirements (Figure 9.1).

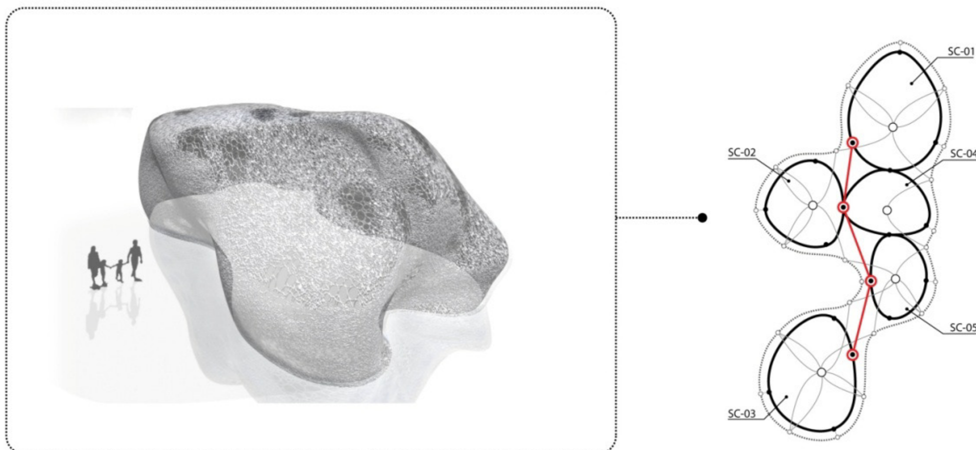


Figure 9.1:

The formalism of the design concept shows five spatial components with interrelations between them wrapped by a skin

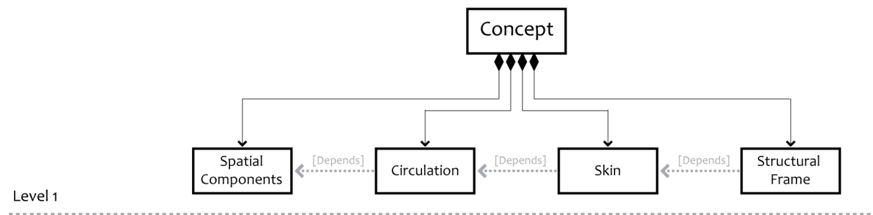
9.1.2 Decomposition

9.1.2.1 Component Decomposition

The design concept will be decomposed initially into four main components. These will be the spatial components, the floors, the skin and the structural framing (figure 9.2). Based on the design concept it is clear that there is a strong dependency between all four components and that all four are integrated within the design process. For example, if a spatial component changes its location, then the floors are affected and the skin is modified which also affects the structural frame. This dependency will have to be taken into account in the aspect decomposition. Furthermore, both the skin and structure will be further decomposed in the subsequent level. The synthesis modules will have to generate these different components.

Figure 9.2:

Component Decomposition

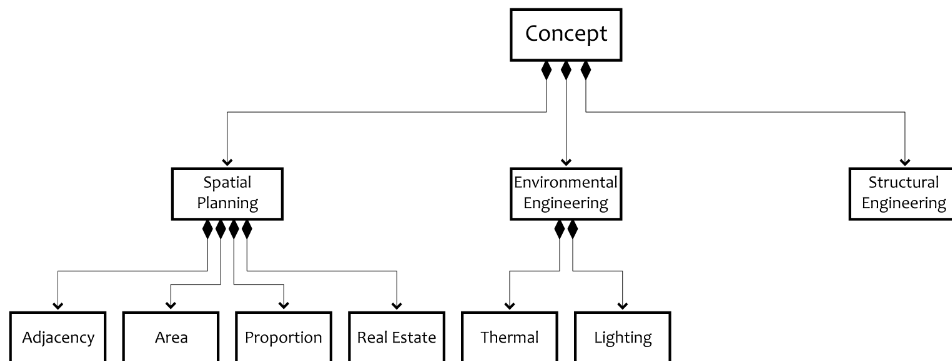


9.1.2.2 Aspect Decomposition

Based on the dependency between the different components in the initial developmental level, the aspects of interest in all four components will have to be identified simultaneously.

Figure 9.3:

Aspect Decomposition



In the case of level one, the aspects will be first decomposed into spatial planning, environmental and structural aspects. Spatial

planning aspects will be further decomposed into adjacency, area, proportion, and real estate aspects. Environmental aspects will be decomposed into thermal and lighting aspects. Structural aspects will not be decomposed further (figure 9.3). These will represent the main aspects that we will analyze for.

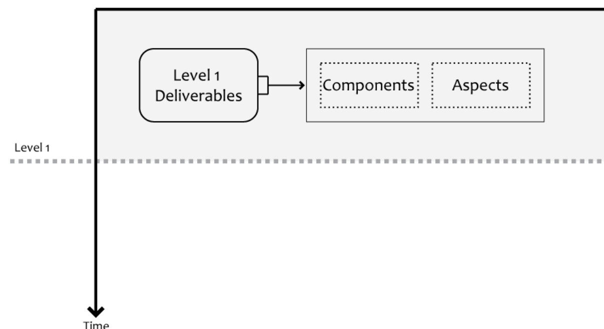
9.1.2.3. Development Decomposition

Within development decomposition decisions have to be made on what will be included in a certain level and what will be left for subsequent levels. This will identify the expected deliverables of a certain level.

Within our current experiment, the deliverables of level one will include a configuration of the spatial components, the floors, the skin and structural frame (figure 9.4). These configurations will have to be assessed for adjacency, area, proportion, real estate, thermal and lighting.

Figure 9.4:

Development decomposition



9.1.2.4. Activity Decomposition

Based on the design concept and the mapping between components and aspects we noticed that structural frame only maps to the structural aspect. For the sake of simplifying the design task, and although the structural frame is dependent on the other components in its design process, the design team made a decision to decouple structure at this level (figure 9.5).

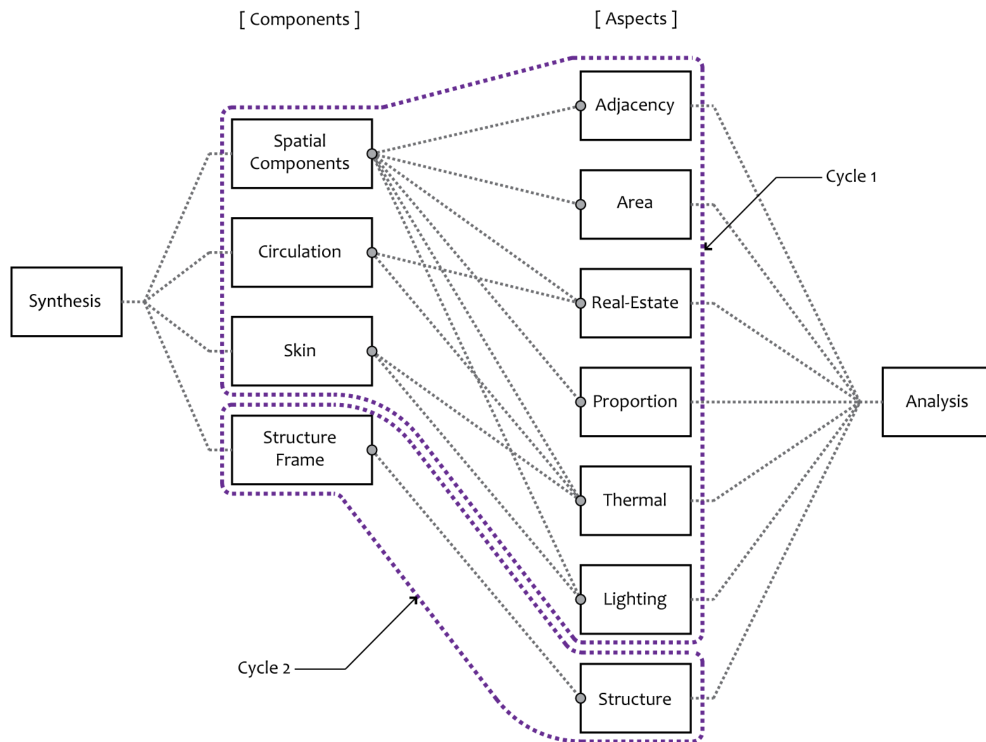
This will generate two design cycles at this level. The first tries to solve the spatial planning aspects as well as the environmental aspects. The second should take the output of this cycle and generate the sizing of the structural system. This experiment will focus on the first cycle.

This design cycle can be further decomposed into design activity modules. These design activity modules will include synthesis,

analysis, evaluation and optimization modules. Given a design vector, the synthesis modules will generate the spatial components, floor, and skin. The analysis modules will analyze for adjacency, area, proportions, real estate, thermal and lighting behaviors. The evaluation modules will aggregate the different behaviors of the different analysis modules into a general performance quantity. Finally, given the outputs of the evaluation modules, the optimization modules will search the design space and specify a new design vector.

Figure 9.5:

Component and aspect decomposition mapping



9.1.3 Formulation

As opposed to the top down decomposition of the design concept into modules, the process of assembling and formulating the current MDDS design cycle is a bottom up approach (figure 9.6).

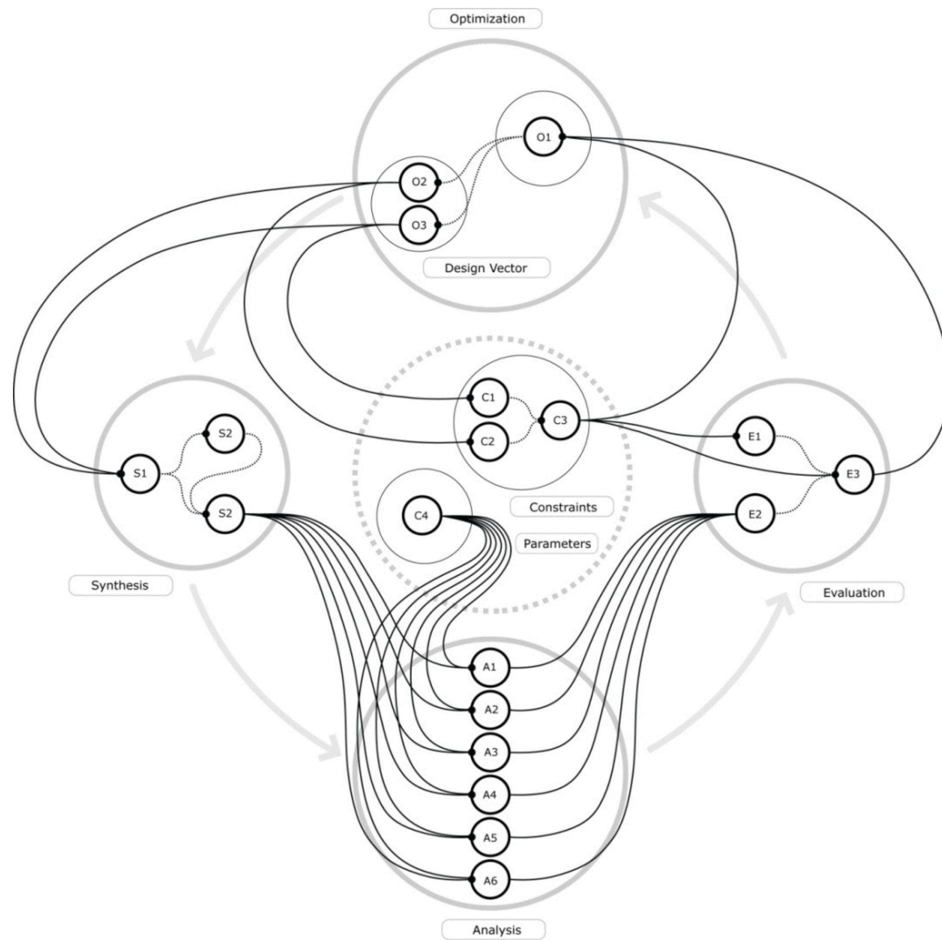
As stated in decomposition, the synthesis modules, given a design vector, should generate the spatial components, floor and skin. This will be achieved by an assembly of three synthesis modules: the rule set module, the data structure module and the inference engine. These three modules will work together as a unit. This synthesis assembly receives the decoded design variables from the design vectors module in the optimization cluster and outputs a design

solution (phenotype) that can be analyzed.

Within the analysis cluster, each of the six analysis modules receives relevant data from the synthesis assembly. Each module should then provide a measurement of the design performance for a certain aspect.

Figure9.6:

The MDDS cycle on level one



The evaluation module controls the flow of data by making sure that those designs that do not comply with the constraints are not sent to the synthesis assembly to be further developed into a full design solution (phenotype). In fact, the data flow path between the synthesis assembly and the analysis cluster represents a bottleneck

within the cycle due to its computational burden. Therefore, it has to be managed efficiently. If the constraints fail there is no need for a phenotype to either be formulated or analyzed, since doing so would only waste computational resources. However, if the constraints are not violated, the performance measurements generated by the different analysis modules are then aggregated into an objective function that acts as a figure of merit. The evaluation modules handle the multi-objective output generated by the different analysis modules.

Due to fact that we are at an early stage of design, a GA heuristic algorithm was chosen as the optimization algorithm. The GA in the optimization assembly will evaluate the fitness of the design solutions in the population. Several solutions will be chosen based on their fitness and undergo genetic transformations to form a new population. The GA runs until satisfactory fitness levels are reached.

9.1.4 Modeling

9.1.4.1 Synthesis Modules

The synthesis assembly will mainly consist of three modules: the data structure module, the rule set module, and the inference engine module. In constructing the geometry of a design solution, a bottom-up approach is taken within the data structure module. Three main data structures are implemented: cells, spatial components and skin. The data structure module is organized hierarchically with feedback loops between the different data structures.

A cell represents the elementary unit of the space in which the spatial components are to be allocated. It is defined by a set of control and boundary points and construction lines. It has knowledge about its location and its neighboring cells. It also knows the status of its occupancy and by which component it is occupied. In this experiment we implemented eight cells. Each cell contains four control points with at least two shared with neighboring cells. There are a total of twenty-two control points.

A spatial component grows in a cell, inheriting all its base geometry. It has a reference to the component object, which in this case is the spatial component spline Furthermore, when the skin is generated, the component knows which skin regions it is associated with. It also has a set of attributes that include perimeter and area of the spatial component, length of each segment of the component region, and the normal angles that define the orientation of each segment.

The skin is generated from the geometric configuration of components. The skin grows sequentially from supports that are generated from both the components and the cells they occupy. The data structure includes a reference to the skin's object, which in this case is the skin profile spline. It also includes an attribute that defines the area enclosed by the skin.

In regards to the rule set, our approach draws from shape grammars pioneered by Stiny and Gips (Stiny and Gips, 1972). Although the spatial relations and dependencies can be coded directly in the grammar, the automation of general shape grammars is difficult due to the recurrent emergence of new shapes in the process. A class of shape grammars that is applicable to computer implementation is *set grammars* (Stiny 1982). Set grammars consider shapes as symbolic objects and therefore do not require difficult sub-shape matching procedures. This is the approach used here. The grammar implemented is based on three fundamental design-rule sets (figure 9.7). These rules draw from knowledge built into the data structures and are organized hierarchically.

The first set of design rules deals with the allocation of spatial components. There are six rules in this set. The second set of design rules deals with the deformation of the spatial components by altering the coordinates of the control-points that define the spatial components in both the x and y directions. There are two rules in this set, one for each coordinate.

The wrapper skin poses considerable difficulty. Since the components configuration boundary is not a convex hull, the skin cannot be formulated using known algorithms for that class of problem. A possible approach to solve the problem would be by using a local optimization loop. However, this would create an extra layer of complexity that would be computationally exhaustive. An alternative approach would be using a set of parametric rules that can generate the skin directly. This is the approach assumed here. These rules compose the third rule set.

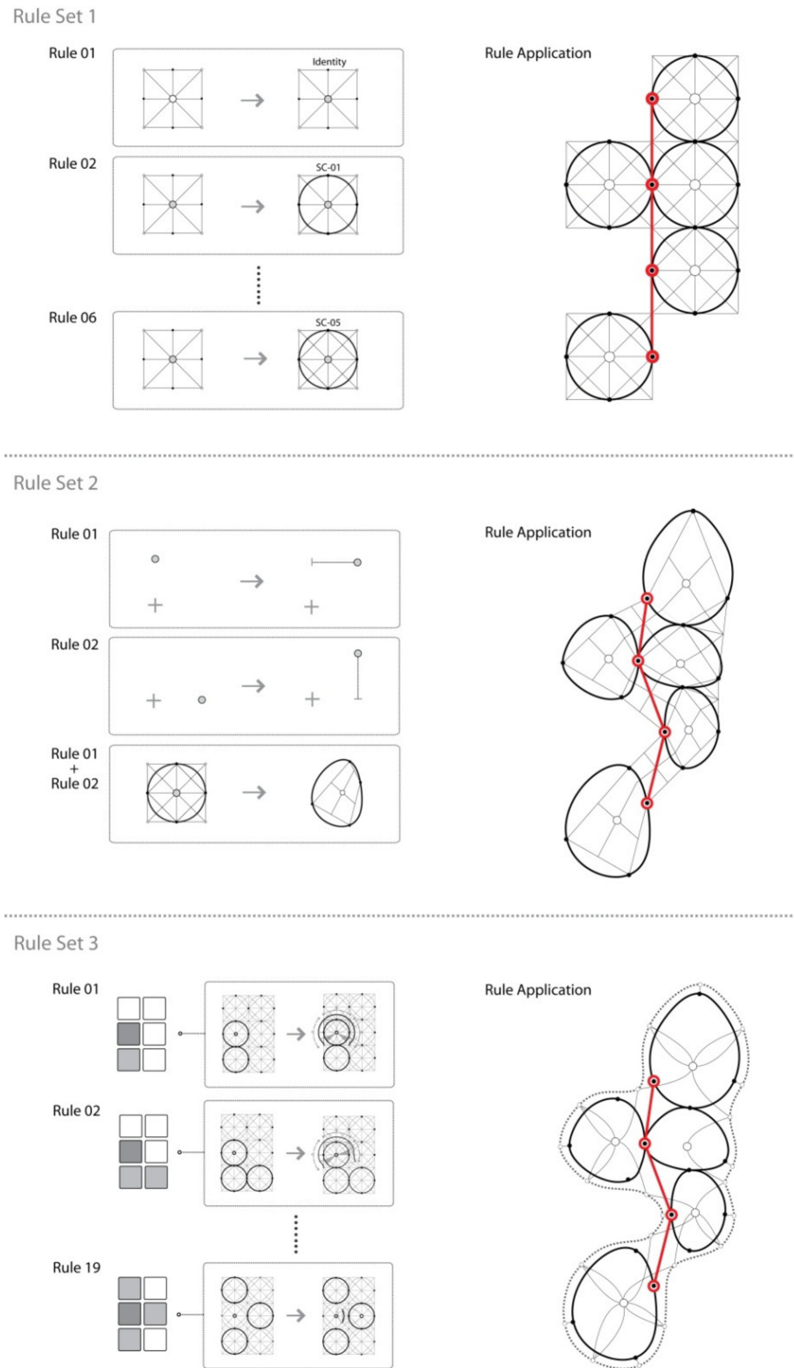
Each rule in the third rule set is applied locally to each cell and each component in a counterclockwise fashion generating the skin supports from which the skin grows. The skin starting and ending control points are based on the cell-underlying grid. There are nineteen rules in this set that can capture all the different generated configurations. Each rule divides the skin into *regions*. This partition of the skin is a useful representation for many of the analysis modules implemented, since it determines each component exposed regions in an additive piecewise manner. Each component region is in turn broken down into segments. Depending on the rule applied

the number of these segments range from one to four (figure 9.8).

The synthesis grammar rule sets implemented here are fundamental operators that cannot be decomposed or recomposed. The rule sets contain all required rules and the aim of the generative mechanism is to find satisfactory sequences of these rules.

Figure 9.7:

Three rule sets define the synthesis grammar.



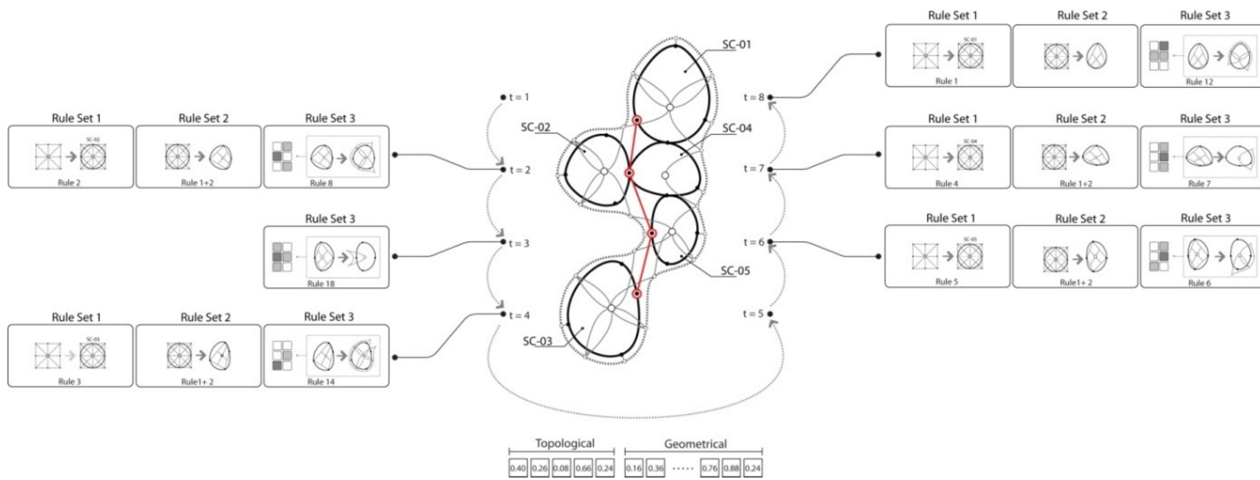


Figure 9.8:

The sequence of application of the three rule sets. Starting with the first cell at time t =0 and ending with the last cell at time t =8.

The design vector that provides the inputs to the synthesis phase is divided into two types of variables, namely topological and geometrical. These topological and geometrical variables have generally been implemented separately in space planning problems. They are handled within the synthesis phase by the first and second rule sets respectively. The third rule set builds on the outcome of the first and second rule sets.

The inference engine scheduler applies the rule sets sequentially in an orderly manner. The interpreter searches each rule set for the matching rules of the current state and fires them when appropriate. The rules of the third rule set are context sensitive and function like a simple two dimensional cellular automata that analyses each neighbor's occupancy and decides which rule to apply. All three synthesis modules were implemented in the CATIA VBA environment.

To summarize, the inputs to the synthesis modules are:
a- Location of the spatial components which handles the topological variables
b- Location of the control points in the system which handles the geometrical variables

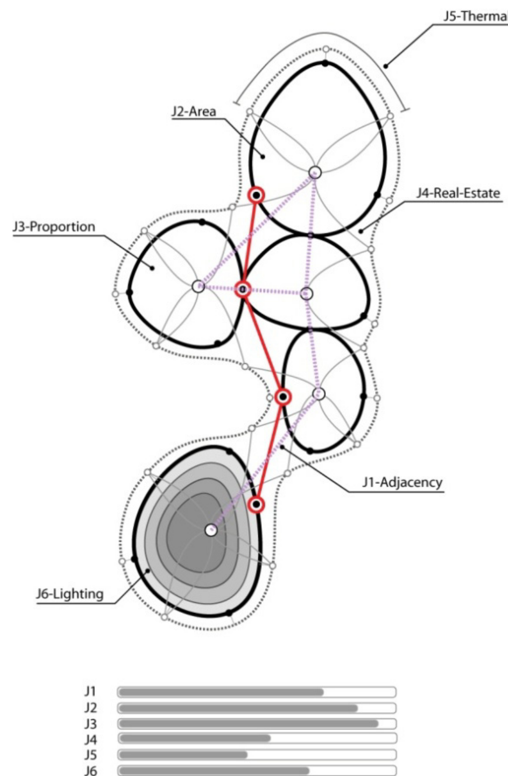
And the outputs are:
a- Area of each spatial component
b- Perimeter of each spatial component
c- Length of each spatial component region
d- Length of each segment composing a region
e- Orientation of each segment composing a region
f- Total area enclosed by skin

9.1.4.2 Analysis Modules

In the proposed experiment, the design concept will be broken down into multiple single-disciplinary analysis modules in order to evaluate how well it performs from the point of view of each discipline separately. These modules include: an adjacency module, an area module, a real estate module, a proportion module, a thermal module and a lighting module (Figure 9.9).

Figure 9.9:

The analysis phase includes six analysis modules.



Since we are working at the design concept stage, the level of detail of the overall design constitutes a simplification of reality. Any rigorous analysis may go beyond the scope and the precision of the overall design description. Therefore, the models we will use for the different discipline modules will be based on heuristics or simplified representations to test the feasibility of design solutions. The modules are implemented in VB Scripts or Excel and built in VBA Scripts.

Adjacency Module

A functional rationale determines the adjacency requirements that the spatial components have to comply with. These requirements are treated and quantified as a set of “bond forces” that tie together

all components pair-wise. In the adjacency module, the actual design–in terms of the location of the spatial components–is examined and rated against these requirements.

An example of a set of adjacency requirements is given in table 9.1 (a higher adjacency attraction corresponds to a higher number).

Table 9.1:

Adjacency requirements.

	SC-01	SC-02	SC-03	SC-04	SC-05
SC-01		0	2	1	0
SC-02	0		0	1	1
SC-03	2	0		0	0
SC-04	1	1	0		1
SC-05	0	1	0	1	

Given a spatial component *A*, the module considers its "neighbours" and checks whether any of those is associated to *A* in the adjacency table, and how strong the intensity of the bond is. That figure is then multiplied by a geometric coefficient that quantifies how "close" the neighbours are, according to table 9.1.

As the above figure shows, this filter strongly amplifies the score of the N-S neighbourhood relationship; E-W adjacency comes second since it involves the crossing of the circulation spine, being amplified by a factor two, then at last any diagonal relationships are considered. Any more distant are not considered. In symbolic terms, the rating for one component is given by:

$$J_{adja,i} = \sum_{j=1}^{N} a_{ij} g_{ij} \ ,$$

where

a_{ij} is the element in the ith row and j^{th} column of the adjacency matrix (Table 9.1), and

g_{ij} represents the geometric coefficient given in figure 9.10, when component i lies at the centre of the diagram and component j.

The total output is given by:

$$J_{adja} = \sum_{i=1}^{N} J_{adja,i} + J_{pena}$$

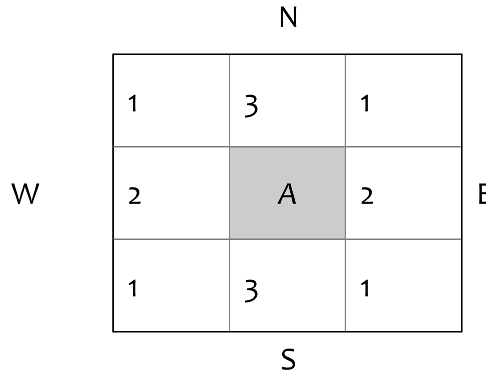
Where

$J_{adja,i}$ are the output from each component and

J_{pena} is a penalty that applies if the entrance is not on the West side of the building (were the street is supposed to be).

Figure 9.10:

Adjacency score amplifying factors *g*.



Area Module

The area module compares the areas of the spatial components generated by the synthesis phase with the areas prescribed by the architectural area program. It favours solutions with a high compliance with the program and flags solutions that show a worse compliance.

For each spatial component, the following function is calculated:

$$J_{area,i} = \min\left(\frac{A_{req}}{A_{act}}, \frac{A_{act}}{A_{req}}\right)_{,i}$$

$$J_{area,i} \in (0;1)$$

Where:

A_{req} is the area of the ith spatial component, as specified by the program, and

A_{act} is the value corresponding to the actual design.

The values from all spatial components are then averaged to obtain an overall value:

$$J_{area} = \frac{1}{N}\sum_{i=1}^{N} J_{area,i}$$

J_{area} lies in the range (0,1); an optimal design corresponds to an output of 1, whereas lower values flag solutions that show a worse compliance with the program.

Real estate Module

This module compares the floor plan's net area to its gross area. It aims at minimizing the space between the spatial components. These are areas that are allocated to circulation, but have a lower real estate value.

The circulation area is calculated as:

$$A_{circ} = A_{tot} - \sum_{i=1}^{N} A_i$$

Where

A_{tot} is the total floor area enclosed within the skin boundary, and

A_i is the area of the ith spatial component.

The scalar:

$$J_{circ} = 1 - \left(\frac{A_{circ}}{A_{tot}}\right)$$

is output to the performance module; it ranges in the interval (0;1), the higher values corresponding to a more favourable design.

Proportion Module

The shape of the spatial components is determined by the location of the control points. A spatial component may have a multitude of possible shapes due to the location of the control points. This module aims at promoting more skewed or slanted shapes that produce aesthetically more appealing layouts, while keeping elongation and distortions within acceptable limits. This module filters any regular or fairly irregular shapes and discourages highly irregular forms.

The proportions module was designed to quantify and rate this aspect of the design. The ratio:

$$e = 4\pi \frac{A}{p^2}$$

was elected as the numerical benchmark for proportion. In fact, for all closed curves, the *Isoperimetric Inequality* states that $e \leq 1$, the equality exclusively holding for the circle. Lower values of *e* correspond to more slanted or oblong curves. As such, the output function of the module was built to filter with a higher rating any regular or fairly irregular shapes and to discourage highly irregular forms, according to the following law:

$$J_{prop,i} = \left[\min\left(1, \frac{e}{e_0}\right) \right]^{\alpha}$$

where e_0 and α are shape parameters that control the width of the plateau and the slope of the descending branch, respectively. The average of the above output functions, as there is one per spatial component, is sent to the performance module:

$$J_{prop} = \frac{1}{N} \sum_{i=1}^{N} J_{prop,i}$$

Thermal Module

Assuming that we are building in a cold climate, it is important to provide a design that minimizes thermal losses and favoured solar gain. A simple thermal module was devised to measure the energy balance. A few simple assumptions were adopted, due to the minimalism of the design model. Yet these simple assumptions proved capable of capturing the fundamental relationship between the shape of a building and its environmental performance.

The assumptions that were used in the module include:

- The heat exchange is duo-dimensional or, equivalently, the material properties of the cladding materials do not vary along the floor height.

- Heat exchange solely takes place in the exposed regions of the spatial components' perimeter, as these are possible fenestration locations. This crude but reasonable assumption neglects the energy dispersed through the walls, as this is usually considerably smaller than the heat flow lost through the windows panels.

- Solar gain depends on the local orientation of the façade, being maximum on the South side and negligible on the North side. Its maximum relative intensity–with respect to the conductive and convective mechanism–is a design parameter, and can be varied according to the intended location of the house.

According to these assumptions, a simple algorithm computes the heat exchange balance:

$$Q = \sum_{j=1}^{R} \left(L_j k_{q,j} \right)$$

where

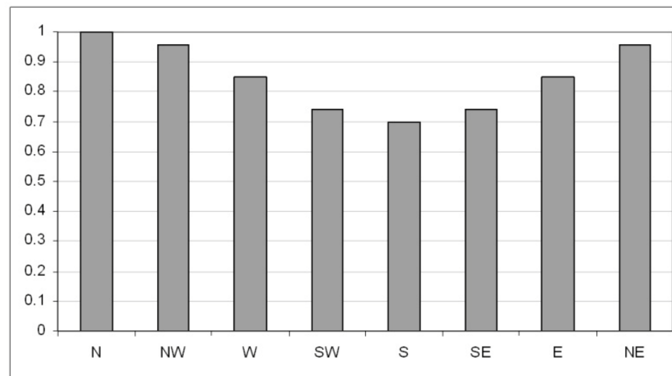
R is the number of exposed regions,

L_j is the length of the jth exposed region and

$k_{q,j}$ is a parameter that defines the normalized heat loss per unit length of the region. $k_{q,j}$ depends on the average orientation of the region and includes both conduction and convection and solar gain. The relative importance of these two mechanisms is determined by a parameter and can hence be adapted to the geographical, climatic and technological conditions. k_q varies as shown in Figure 9.11.

Figure 9.11:

Variation of k_q (and k_L) with orientation.



The total heat loss Q is then normalized with respect to a characteristic length of the building, obtained as the square root of the total floor area:

$$\overline{Q} = \frac{1}{\sqrt{A_{tot}}} Q$$

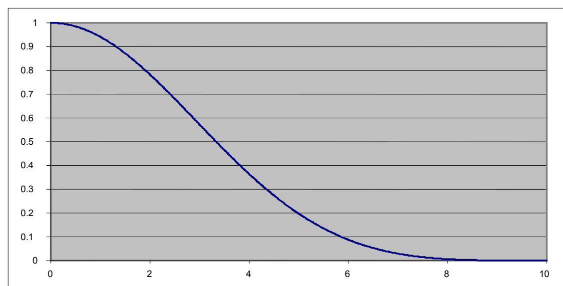
The most advantageous design in terms of heat loss would generate a minimum normalized loss $\overline{Q}$ = 0 (no heat loss). Higher scores correspond to solutions with a worse balance. The final module output J_{ther} is a transform of $\overline{Q}$ according to a monotonic function:

$$J_{ther} = J_{ther}\left(\overline{Q}\right)$$

Note that J_{ther} needs to be maximized, and its optimum value corresponds to 1. Its graph is shown in Figure 9.12.

Figure 9.12:

The function $J_{ther}(q)$.



Lighting Module

The day lighting performance is assessed by adopting a simple geometric model. The module measures, for each exposed region in a spatial component, the fraction of the area that is exposed to sunlight and multiplies it by a coefficient that depends on orientation. A number of physical simplifications were also adopted.

$$A_{lit,\mathrm{mod},j} = A_{lit,j} k_{L,j}$$

For each region, the illuminated fraction $A_{lit,i}$ of the total component area is the area of the curvilinear figure bounded by the exposed region, its offset at a distance d toward the centre of the component. The coefficients $k_{L,i}$ depend on the average orientation of the region, according to figure 10.

The variation of k_L against orientation is given by a sinusoidal law whose maximum (k_L = 1) corresponds to the South and its minimum ($k_L = k_{L,min}$) to the North:

$$k_L \in \left[k_{L,\min};1\right] \qquad [*]$$

$k_{L,min}$ is a parameter that can be modified according to the location of

the building and varies with orientation. The total value is then calculated:

$$A_{lit,\mathrm{mod}} = \sum_{j=1}^{R} A_{lit,\mathrm{mod},i} \qquad [**]$$

Note that if $k_{e,j} = 1 \;\forall j$ then $A_{lit,mod}$ would physically represent the total illuminated area. Otherwise, the k_e coefficients act as a filter and decrease the scores of the regions that do not face South. $A_{lit,mod}$ is then divided by the total physical floor area to obtain the output J_{lite}:

$$J_{lite} = \frac{A_{lit,\mathrm{mod}}}{A_{tot}} \qquad [***]$$

It follows from ***[*], [**]*** and ***[***]*** that $0 \leq J_{lite} < 1$, unity corresponding to an optimum performance.

Constraints Modules

There are two main constraint modules implemented. The first is a topological constraint module; and the second is a geometric constraint module. Both modules act on the design vectors and not on the design solution generated from the synthesis phase.

In the topological constraints module the spatial components are tested against the adjacency requirements specified by the designer. Two components may have a strong bond, a weak bond or no bond at all. The solution generated by the synthesis assembly may or may not comply with the requirements, and, in particular, some of the strong-bond relationships may not be obeyed. The topological constraints ensure that the number of violated strong-bond relationships does not exceed a pre-set threshold.

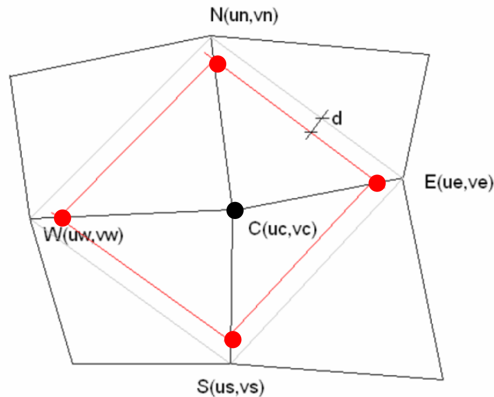
Although the design vector module implemented in the optimization phase constrains implicitly the coordinates of the control points to lie in a predefined order within the row or column they belong to, this condition is not sufficient to keep the deformation of the grid within acceptable limits. The geometric module handles the deformations and prevents any excessive distortion of the grid that might create non-convex spatial components.

In addition to the constraints modules, a constants and data storage module was also implemented. This module contains all the constants and parameters used by the different modules such as location, climate, and area program among others.

In order to avoid any excessive distortion of the grid, each control point must lie within a specific region, delimited by the red lines in figure 9.13. Each red line is obtained by offsetting the corresponding gray line by an amount *d*.

Figure 9.13:

Graphical representation of the geometric constraints.



These constraints prevent each quadrilateral region from assuming a non-convex, hourglass-like shape, resulting in computational problems and in an unacceptable design. This kind of constraint is defined by the following equations (in the case of quadrant NE):

$$\begin{vmatrix} v_C & u_C & 1 \\ v_N & u_N & 1 \\ v_E & u_E & 1 \end{vmatrix} > \frac{d}{\sqrt{(v_N - v_E)^2 + (u_N - u_E)^2}}$$

Where:

u_c, v_c are the coordinate of the point subject to the constraint;

u_E, u_N, and v_E, v_N are the coordinate of the adjacent points, respectively to the East and North of point C.

9.1.4.3 Evaluation Modules

There are three evaluation modules implemented. The first is a *flow control module* that evaluates if the design vector violates the constraint modules in the analysis cluster. It acts as a switch directing the data flow to either of the other two evaluation modules. The other two modules are the *feasible design* and *infeasible design* modules. The flow control module triggers the infeasible design evaluation module if the constraints are severely violated. If the constraints are not violated, the flow control module triggers the

feasible design evaluation module.

Although the constraints are handled by the optimization modules, the flow control module is important from a design-process management point of view. If the design vector is infeasible the flow control module would bypass the synthesis and analysis modules saving extensive computational time.

The infeasible design module simply signals the violation to the optimization modules and ranks the design solution in proportion to the number of violated constraints.

The feasible design evaluation module on the other hand triggers the synthesis and analysis modules. All the ratings (J_{area}, J_{circ}, etc.) of the disciplinary performances that originate from the analysis modules converge into the feasible design evaluation module, where they are aggregated to generate an overall evaluation of the design, according to the standard *scalarization* approach.

The final multi-disciplinary performance J is the weighted average of the normalized output from the various modules:

$$J = \frac{\sum_{m=1}^{M} w_m X_m}{\sum_{m=1}^{M} w_m}$$

where

$M = 6$ is the total number of analysis modules;

X_m is the normalized output from the mth analysis module, and w_m denotes the corresponding weight. In particular, X_m is obtained by normalizing the actual output of the mth module J_m according to:

$$X_m = \frac{J_m - J_{m,\min}}{J_{m,\max} - J_{m,\min}}$$

By adopting scalarization, it becomes straightforward and convenient to explore the influence of one or more disciplines on the overall design by amplifying the corresponding weights w_m.

9.1.4.4 Optimization Modules

The optimization modules consist of two groups of modules. The first contains the optimization algorithm, and the second is a converter module which converts the outputs of the optimization algorithm into data which the synthesis modules can understand.

The design concept clearly entails a multi-performance space-planning problem. Space-planning problems have been studied by many researchers (Buffa et al, 1964). They are considered one of the most interesting and difficult of formal design problems

The NP-Completeness of the space-planning problem makes it impossible for any process to guarantee finding the optimal solution within a reasonable time. There are no known algorithms for this problem. Its difficulty arises from its complex nonlinear nature and from the combinatorial character of generated solutions (Jo and Gero, 1998).

During the synthesis phase a large number of possible solutions can be generated even with a small number of space components. The number of solutions grows exponentially as the number of space components increases. During the analysis phase, the multiple performance and objective requirements involve expensive computations due to the very large number of solutions to be analyzed.

Design Vectors Modules

The design vector contains the design variables. It guides the design solutions by informing the synthesis modules, like the DNA of an organism. These design variables are produced by the optimizing algorithm, which “learns” from the performance history of the previous generations and tries to find values for the variable that increase its collective performance. Two major categories of design variables have been considered in our experiment and are implemented in two different modules these are: the topological variables module and the geometric variables module.

The topological variables module defines the cell location of each spatial component. Instead of creating constraints that prohibit the allocation of two different spatial components in the same cell, this check is performed implicitly within this module. This guarantees that no two spatial components are placed in the same cell.

The geometric variables module on the other hand guarantees that the control points in a row or a column remain distributed in an

organized manner, in order to minimize singularities while generating the cells.

Genetic Algorithm Module

The optimization algorithm to be implemented has to maximize the multi-disciplinary performance *J* formulated in the evaluation module. Due to the nature of the design space, the search algorithm implemented should not be limited by restrictions of continuity or existence of derivatives. Therefore, a heuristic search algorithm was considered to be suitable for this case. A Genetic Algorithm (GA) was implemented.

Genetic algorithms are modeled after natural evolution, which is able to create a large set of creatures that are suited for their environments. The GA's representation is done at two levels, namely the genotype level and the phenotype level.

The genotype is the implicit representation of an individual design solution. It consists of a sequence of coded instructions analogous to DNA in natural evolution. In our experiment these instructions are of two types: topological instructions for allocating spatial components, and geometric instructions for modifying control points that affect the cells. All the genetic transformations including crossover and mutation happen at the genotype level.

On the other hand, the phenotype is the interpretation of genotype at the physical level. It is the external perceptible representation of the genotype. The behaviors of a design solution can be observed at this level. Therefore, the analysis task is performed to design solutions at this level.

The evolution starts from a population of randomly generated design solutions, in addition to a few seeded acceptable design solutions to guide the evolution. (A successive seeding methodology was implemented; three seeding phases were applied throughout the evolution). Evolution happens in generations. For each generation, the fitness of every design solution in the population is evaluated. Based on the fitness the selection operator shortlists the individuals that will survive and breed to form the future generations. In our experiment this fitness is represented by the multi-disciplinary performance *J* formulated in the evaluation module.

Constraints are implemented through the use of penalty functions. If a solution does not comply with the constraints in the system a penalty is added to the fitness of the design solution according to

the degree of violation.

Due to the existence of multi-objectives, the aim is not to produce a global optimum solution, but rather to direct the evolutionary process to produce populations of good solutions. These solutions would be used to study the tradeoffs between the different objectives.

Topological Design Vector (DV01)

For this experiment we have five spatial components and eight potential cell locations. DV01 has the following form:

$$DV01 = \{dv01{,}1 \quad dv01{,}2 \quad \ldots \quad dv01{,}5\}$$

Every element is a real numbers in the range [0,1], and corresponds to a spatial component. An algorithm turns the scalars into actual locations of the spatial components and stores them in the location vector LV:

$$LV = \{lv{,}i\} \qquad i = 1\text{-}8$$

lv,i can either be null (if the ith location is empty) or hold the name of a spatial component. As such, this algorithm decodes the cryptic information contained in the design vector DV01 (the genome of the design) into a tangible design configuration (the location vector).

Geometric Design Vector (DV02)

DV02 contains all the information that generates the geometry of the design in terms of the positions of the Control Points. DV02 has the following form:

$$DV02 = \{dv02{,}1 \quad dv02{,}2 \quad \ldots \quad dv02{,}58\}$$

All variables *dv02,i* are in the range [0,1]. The coordinates of the control points, summarized in the array CP (22 x 2), are computed from the design variable according to a set of algebraic relationships.

As an example, the following equations determine the x-coordinates of control points 3, 4 and 5:

CP(3,1) = CP03,Xcoord = dv02_04/sum(dv02_04 + dv02_05 + dv02_06 + dv02_07)

CP(4,1) = CP04,Xcoord = dv02_04 + dv02_05/sum(dv02_04 +

dv02_05 + dv02_06 + dv02_07)

CP(5,1) = CP05,Xcoord = dv02_04 + dv02_05 + dv02_06/sum(dv02_04 + dv02_05 + dv02_06 + dv02_07)

Similarly, all other control points are defined. It is important to note that, although the application that transforms DV02 into the array CP is not injective, it is surjective, therefore the design (*phenotype*) associated to a design vector (*genotype*) is uniquely defined.

Genetic Algorithm Module

The GA's parameters used in the experiment were:

Population Size: 24
Maximum Generations: 600
Selection Scheme: Multiple elitist
Preserved Designs: 8
Operator Probabilities
  Discrete Variable Crossover: 1.0
  Discrete Variable Mutation: 0.15
Constraint Tolerance
  Maximum Constraint Margin: 0.05
  Percent Penalty: 0.5
Number of Top Designs Stored: 12
Random Number Seed: 4335

9.1.5 Integration & Exploration

When all the modules discussed earlier have been built and their validity verified, the data flow model of the design system is implemented and the modules are integrated. For the integration ModelCenter from Phoenix Integration was used. As discussed earlier in the integration chapter, ModelCenter encapsulates the different programs using wrappers that provide interfaces that facilitate the integration (figure 9.14). The integration of the different design modules produces a powerful design system that acts as a holistic, structured functional unit, capable of searching the design space for valuable solutions.

The MDDS demonstrated promising results (figure 9.15). It managed to generate interesting solutions to our multi-performance space-planning problem. It managed to improve the overall performance of the design solutions beyond our initial seeded solutions. The design solutions in the final populations tended to be compact in their shape. This implies that the design drivers were mostly the adjacency and real estate modules, which were highly weighted in our objective

function. As had been expected, both the lighting and thermal modules were in clear conflict with each other. Since they were both given identical weights, they tended to balance out each other. The program and proportion modules did not seem to have an obvious effect on the design solutions.

Figure 9.14:

MDDS Module Integration

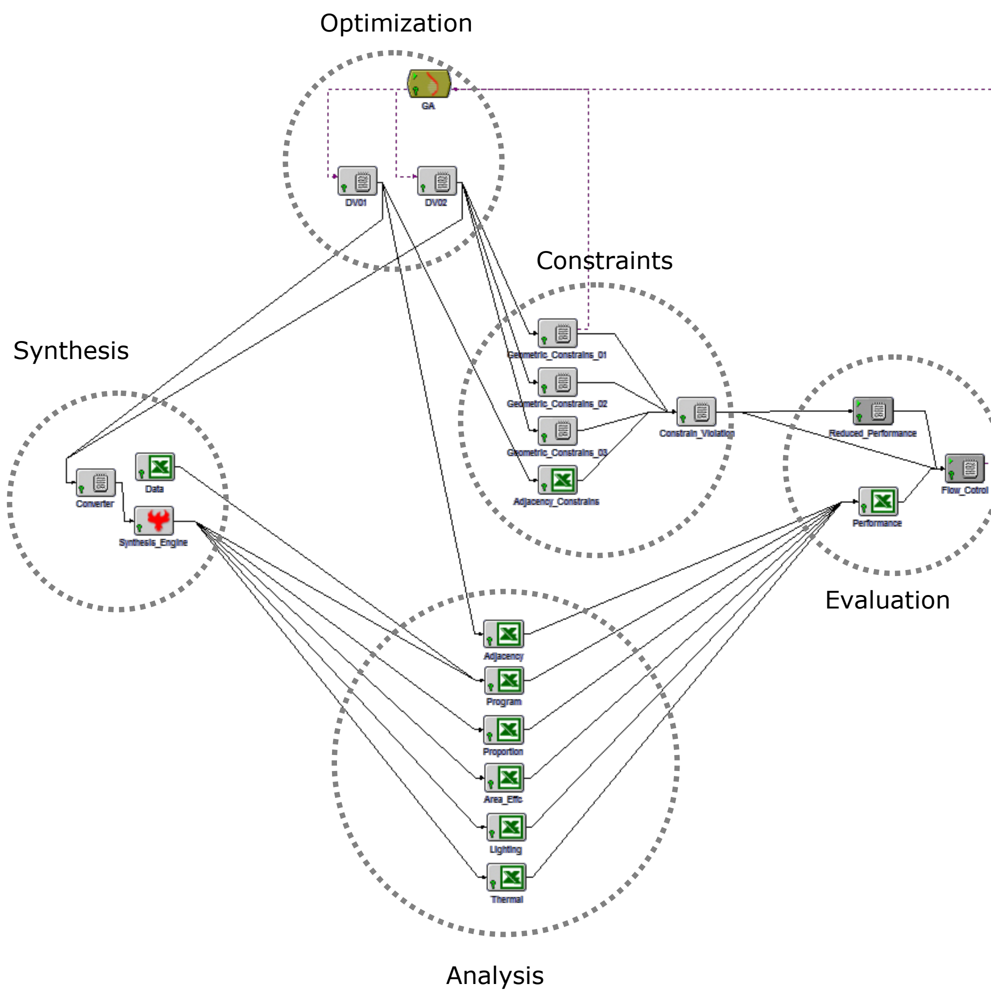
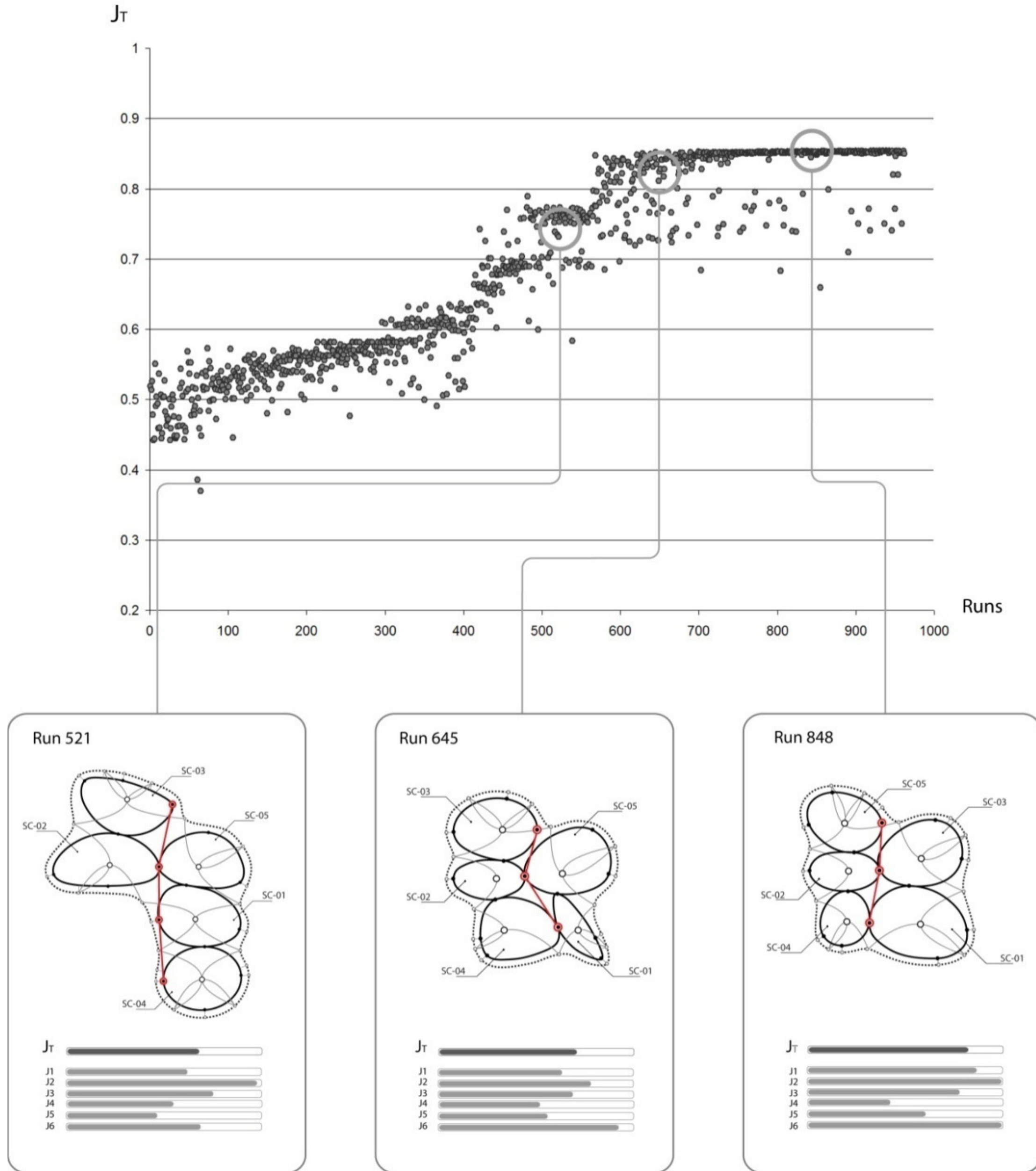


Figure 9.15:

The evolution of solutions. Solutions in the final runs tend to be more compact in their shape.



9.2 Experiment 1 | Level 2

9.2.1 Design Concept

This experiment builds on the output of the previous higher level experiment where the spatial components, floor and building skin were established. This section attempts to present MDDS evolving from the first level into a lower level with more detail. At this level the MDDS will include a new design cycle that will be used to design a building skin.

The design of a building skin is a complex process that involves many disciplines and competencies. The skin is a crucial, active part of the building, because it constitutes its interface with the exterior. It is meant to block or allow the flow of matter, such as rain, people, and energy, in the form of light, heat (or cold) and radiation, following a number of functional criteria. In addition, the skin is the "face" and the "business card" of a building toward the exterior, and must therefore comply with aesthetic requirements and formal equilibrium (or dis-equilibrium).

Therefore, it is evident that the very nature of skin design for buildings is a multidisciplinary one. The skin must meet numerous architectural and technical requirements, such as transparency, sufficient light intake, minimal thermal loss, structural safety, and limited cost of building materials.

The skin in this experiment will be composed of hexagonal cells organized in a "beehive" pattern. Each cell can either host a window or a cladding panel (figure 9.16). It is supported by a two dimensional, non-planar truss of steel pipes, connecting at the nodes in welded joints.

The topology of the "beehive" does not change during the design process, i.e. no new cells can be generated and no cells can disappear, but distances between nodes can vary, and, consequently, the areas of the cells are variable, as well as the lengths of the connecting segments and the amplitude of the angles. The geometry of the grid is described by the spatial coordinates of the nodes.

The nodes are allowed to move on the surface of the façade, generating cells with very different areas and geometry, depending on the requirements and on the constraints. During the design process, the nodes move, driven by the "need" for optimum performance: more light in the interior, less energy loss, increased structural efficiency, architectural preference, etc. Each cell is also

characterized by a material variable, which can assume three states: transparent – corresponding to glass –, semi-transparent (shaded glass) and opaque (cladding panel).

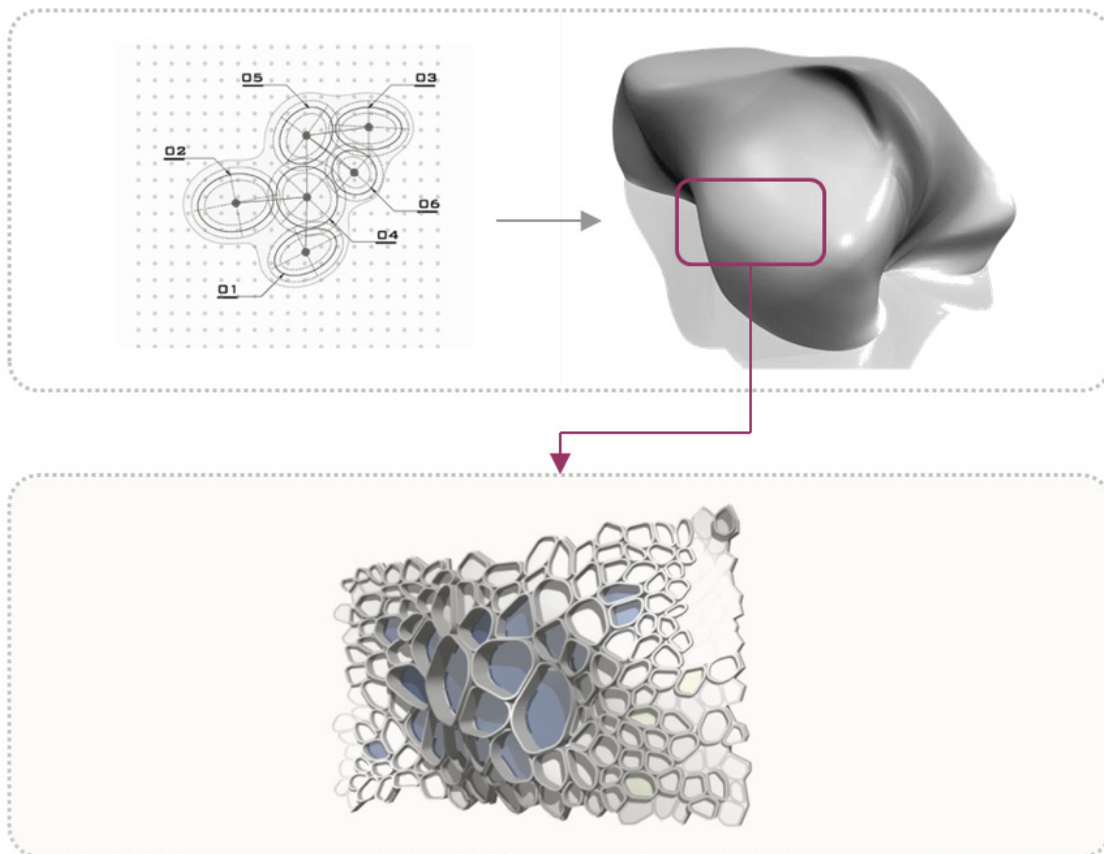
During the design process, the need for more light in the interior and for a lower heat loss through the skin forces the cells to turn transparent or opaque. The goal of the design process is to determine the spatial form of the façade and the material that each cell will be made of.

A MDDS was developed, capable of optimizing the design of the skin on the basis of a lighting analysis of the interior, a thermal analysis of the cooling loads corresponding to the skin configuration, and a finite elements analysis of the supporting structure. The system also considers the architectural need for transparency in the façade due to view requirements of the occupants, and the cost of cladding materials.

The system was developed in three phases/experiments, with more complexity and variation being added to every new experiment.

Figure 9.16:

CAD models of the design concept



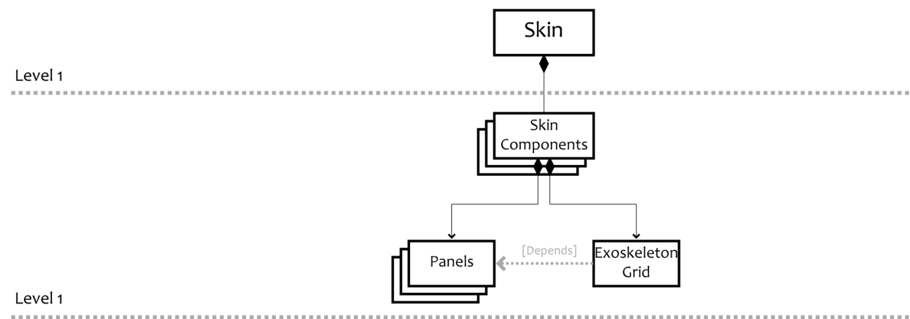
9.2.2 Decomposition

9.2.2.1 Component Decomposition

In this experiment the focus will be on the skin component. Based on the design concept the skin is comprised of several skin components. As mentioned earlier in the previous higher-level experiment, each spatial component has one or more regions that define its interface with the outside environment. Each region is represented with a skin component. Each skin component is composed of two subcomponents. These are an exoskeleton grid and a number of skin panels (figure 9.17). There is a dependency between both components. If the exoskeleton changes size, the panels also change their size. This will have to be taken into account in the aspect decomposition.

Figure 9.17:

Component decomposition

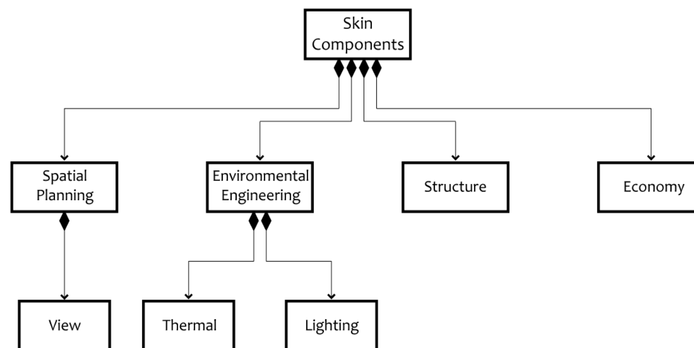


9.2.2.2 Aspect Decomposition

Based on the dependency between the subcomponents of the skin component it will have to be treated as one entity in the aspect decomposition. Therefore, the skin component will be decomposed into five aspects. These are architecture (view requirements), lighting, thermal, structure, and economy (figure 9.18).

Figure 9.18:

Aspect decomposition

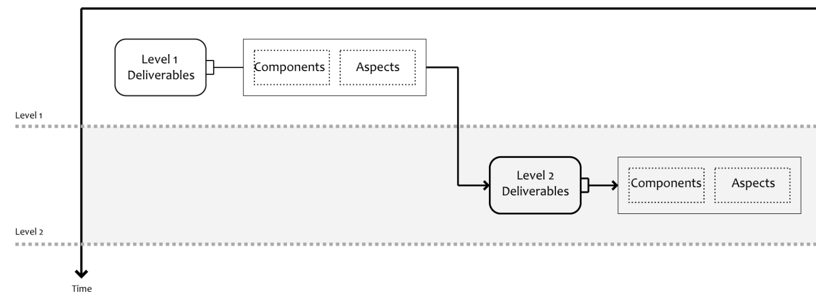


9.2.2.3 Development Decomposition

In this second level of the MDDS, the deliverables will be skin component configuration and structural member sizing. These two deliverables are decoupled in the proposed design and therefore can be implemented in parallel. The focus in this experiment will only be on the skin component (figure 9.19).

Figure 9.19:

Development decomposition



9.2.2.4 Activity Decomposition

Based on the design concept and the mapping between components and aspects we notice that both are fully integrated and coupled (figure 9.20). Therefore, one MDDS cycle will be implemented. It will be expected to determine the spatial form of the façade and the material that each cell will be made of on the basis of a lighting analysis of the interior, a thermal analysis of the cooling loads corresponding to the skin configuration, and of a finite elements analysis of the supporting structure. The system also should consider the architectural need for transparency in the façade due to view requirements of the occupants, and the cost of cladding materials (figure 9.21).

Figure 9.20:

Component and aspect decomposition mapping

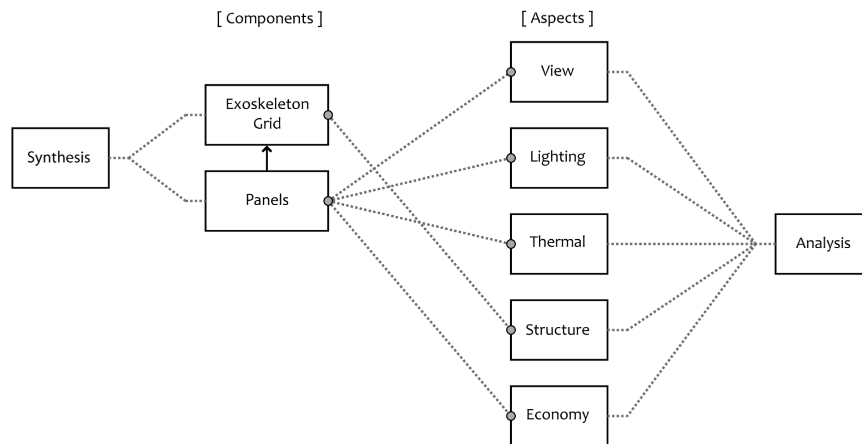
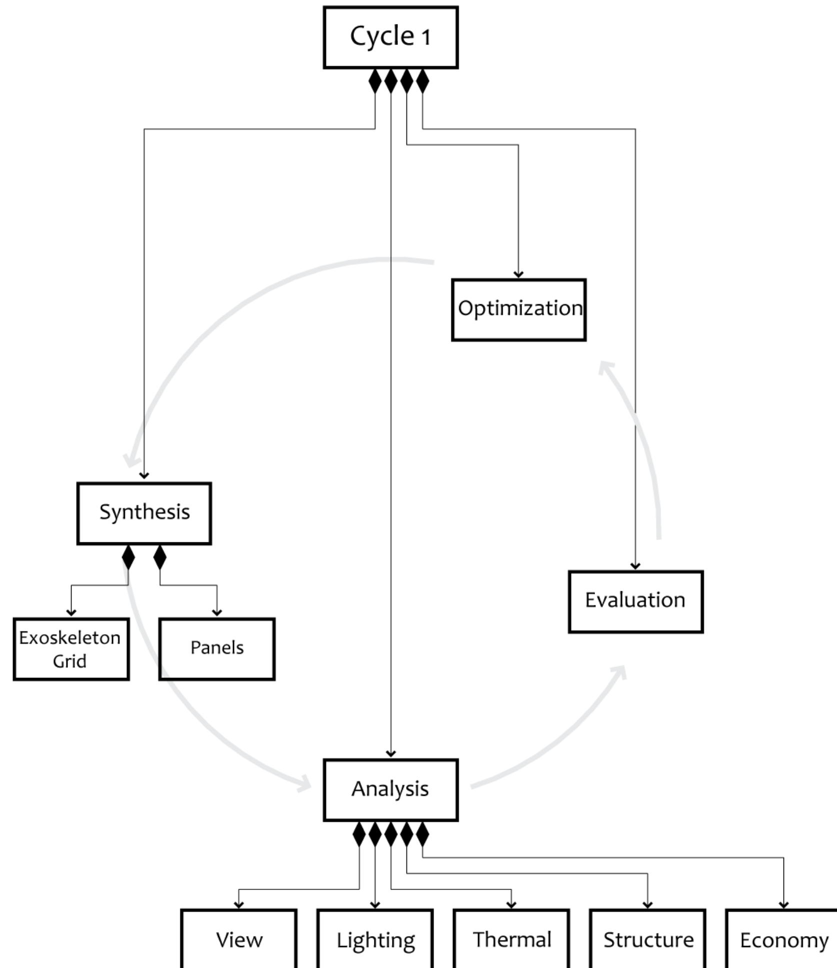


Figure 9.21:

Design cycle one and its design activities



9.2.3 Formulation

A N2 Diagram of the design activity modules was developed, following a disciplinary breakdown (Figure 9.22). The architecture of the cycle was grouped into four clusters, namely synthesis, analysis, evaluation and optimization (Figure 9.23).

The synthesis assembly includes geometry and material modules. The analysis cluster includes the five analysis modules (lighting, thermal, architecture, structures and economy). The evaluation assembly includes modules for the assessment of the overall performance. Finally, the optimization assembly includes the optimization algorithms. All the assemblies are described in the following paragraphs.

Figure 9.22:

Structure of the N² Diagram

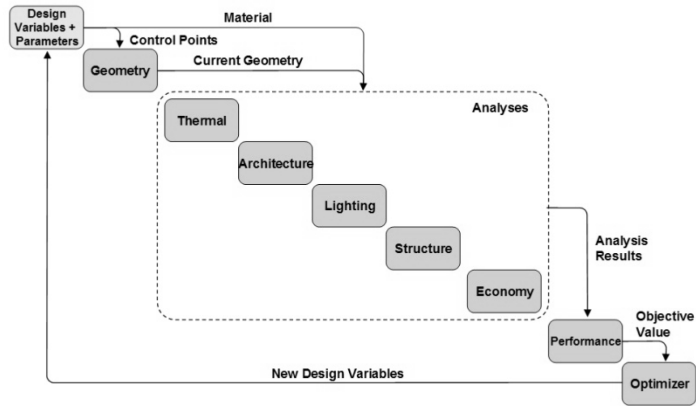
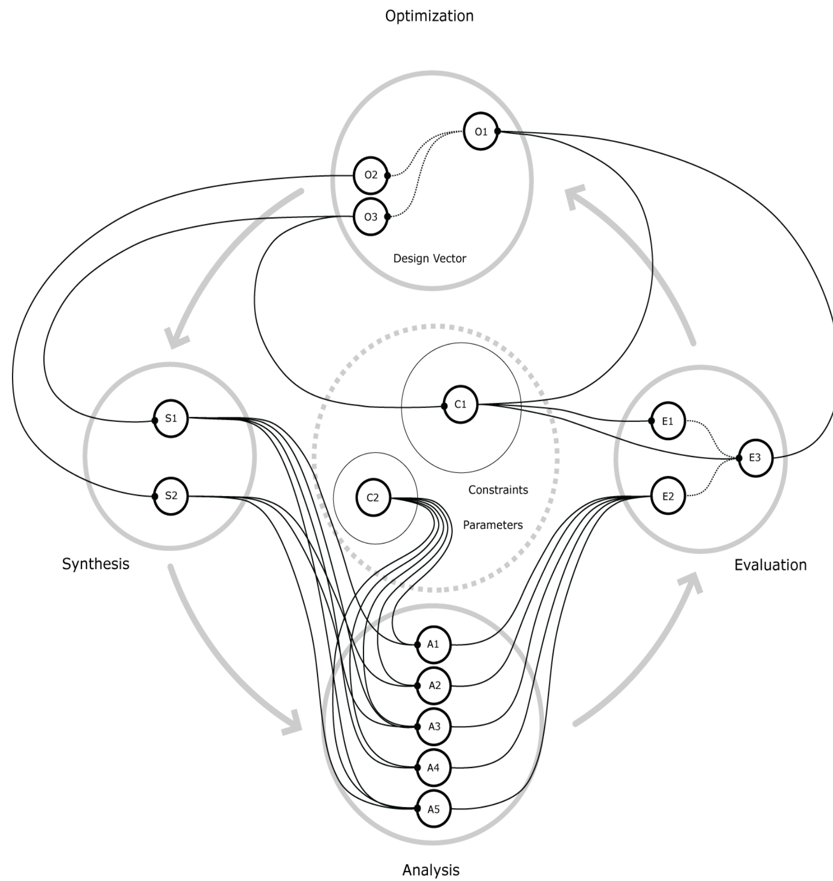


Figure 9.23:

The MDDS cycle on level two



Synthesis	Analysis	Evaluation	Optimization	Constraints & Parameters
S1 Grid Geometry	A1 View	E1 Flow Control	O1 Genetic Algorithm	C1 Geometric Constraints
S2 Material	A2 Lighting	E2 Infeasable-Eval	O2 Geometric DV	C2 Data Storage
	A3 Thermal	E3 Feasable-Eval	O3 Material DV	
	A4 Structure			
	A5 Economy			

9.2.4 Modeling

9.2.4.1 Synthesis

The portion that this study focuses upon is a rectangular surface 5 m wide and 3 m high of our building skin. This portion of the skin is for a building located in Boston, MA, U.S.A. – latitude 42° N, and is oriented towards the South. The Skin represents the interface of a spatial component 6 m deep. The maximum out/inward deflection of the skin was kept at 0.5 m.

Design Vector

The design vector is composed of two main groups of design variables: the geometric and the material variables.

The geometry of the façade is completely described by its nodes. There are 98 or 242 nodes, depending on the experiment, their positions being defined by three Cartesian coordinates (X, Y, and Z) in space, and by "deformed" coordinates on the skin (u, v) and in the perpendicular direction (w).

Due to the constraints on the displacement of the nodes, the Jacobian of the transformation between the two sets of coordinates (X, Y) and (u, v) is always nonzero, i.e. the transformation is non-singular.

Given the computational complexity, only a subset of the nodes was chosen to describe the geometry of the skin. These points are called control points (Figure 9.24). The displacement of each point on the grid is uniquely defined by the position of the control points.

There are 32 Control Points, 16 interior Control Points (four rows and four columns), plus another 16 on the borders. The geometry is therefore fully described by 48 design variables (2 coordinates [u, v] per control point plus 1 coordinate per CP on the border).

The Geometry module, developed in CATIA™ of Dasault Systems, is responsible for the construction of the skin geometry.

The coordinates of the control points are passed to CATIA, which then calculates the positions of all nodes via parametric synthesis model. It also produces the measures of the cells' areas and all other geometric, parametrically-defined properties of the façade.

In addition, CATIA parametrically generates all the structural and non-structural façade components, such as the shading devices, the

steel pipes, the joints, etc., and renders the design.

The second set of design Variables consists of the cell materials. Each cell can be made of glass, shaded glass, or opaque cladding. These materials correspond to three degrees of transparency. The degree of transparency of a cell determines its permeability to light and to heat.

The distribution of materials on the skin is therefore described by 100 discrete variables, which can assume 3 states each (Figure 9.24). As a consequence, the design is controlled by a total of 132 variables.

Figure 9.24:

The distribution of materials and control Points on the Skin

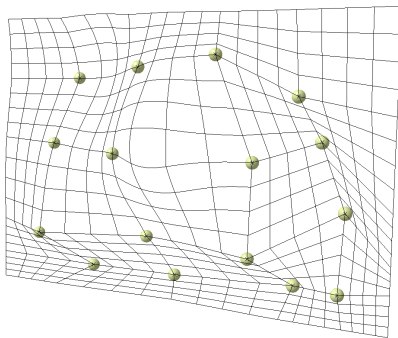
Material Design Vector

3	3	3	3	3	3	2	3	3	3
2	2	2	2	2	2	3	3	3	3
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
2	2	1	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

$$\mathrm{x}_m = \begin{pmatrix} x_{m\,1,1} & x_{m\,2,1} & \dots & x_{m\,i,1} \\ x_{m\,1,2} & x_{m\,2,2} & \dots & x_{m\,i,2} \\ \dots & \dots & \dots & \dots \\ x_{m\,1,j} & x_{m\,2,j} & \dots & x_{m\,i,j} \end{pmatrix}$$

$$x_m \in \{1,2,3\}\ ;\ i,\ j = 1..10$$

Geometric Design Vector



$$\vec{\mathrm{x}}_g = \begin{pmatrix} \vec{x}_{g\,1,1} & \vec{x}_{g\,2,1} & \dots & \vec{x}_{g\,i,1} \\ \vec{x}_{g\,1,2} & \vec{x}_{g\,2,2} & \dots & \vec{x}_{g\,i,2} \\ \dots & \dots & \dots & \dots \\ \vec{x}_{g\,1,j} & \vec{x}_{g\,2,j} & \dots & \vec{x}_{g\,i,j} \end{pmatrix}$$

$$\vec{x}_g \in \{0..1\}^3\ ;\ i, j = 1..6$$

Constraints

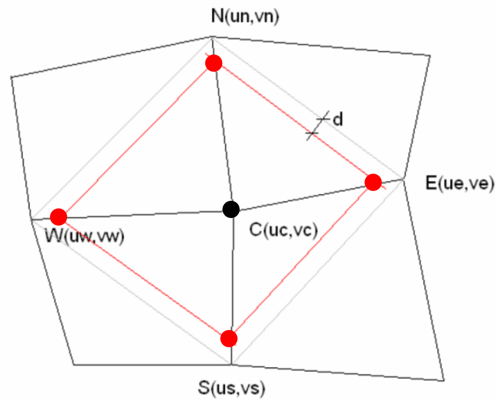
There are several types of constraints that were built within the model.

Firstly, each control point must lie in a specific region, delimited by

red lines in figure 9.25. Each red line is obtained by offsetting the corresponding gray line by an amount *d*. These constraints prevent each quadrilateral region from assuming a non-convex shape, a fact that would create computational problems and would result in unaesthetic design.

Figure 9.25:

Graphical representation of the geometric constraints



This kind of constraint is defined by the following equations (in the case of quadrant NE):

$$\begin{vmatrix} v_C & u_C & 1 \\ v_N & u_N & 1 \\ v_E & u_E & 1 \end{vmatrix} > \frac{d}{\sqrt{(v_N - v_E)^2 + (u_N - u_E)^2}}$$

where:

u_c, v_c are the coordinates of the point subject to the constraint

$u_{E,N}$, and $v_{E,N}$ are the coordinate of the adjacent points that lies respectively to the East and North of point C

Thus, there are 64 constraints of the first type (*geometric constraints*), considering the control points on the borders. Each of these constraints only applies to the correspondent geometric design variable.

Secondly, a constraint on the overall illuminance sets E_{i70} > 300 lux, providing a lower bound for the amount of light measured at 70% of the room depth. Note that this *illuminance constraint* affects all the design variables with different sensitivity.

9.2.4.2 Analysis

A brief description of the analysis modules is provided in the following paragraphs.

Lighting Module

This module calculates the actual illuminance at 70% of the room depth, using the IESNA method, and based on four reference dates of the year.

The Coefficient of Utilization is used to implement an approximation function for better performance (maximum error ±10%):

$$CU_k = (0.362 \cdot RW^3 - 5.98 \cdot RW^2 + 33.1 \cdot RW + 0.0253) \cdot 0.0107 \cdot RR - 1.49$$
$$CU_g = (0.26 \cdot RW^3 - 4.1 \cdot RW^2 + 21 \cdot RW + 1.55) \cdot 0.0093 \cdot w_RR - 1.28$$

With:

$$RR = \frac{room_depth}{window_height}$$

$$RW = \frac{window_width}{window_height}$$

$CU_{k,g}$ Coefficients of Utilization for lighting calculation (sky, ground)

The illuminance for each single cell is calculated as follows:

$$E_{i70} = (CU_k \cdot E_{xvk} + CU_g \cdot E_{xgk}) \cdot \tau$$

where:

E_{i70}	Illuminance indoor at 70% of the room depth [lux]
E_{xvk}, E_{xgk}	Illuminance outdoor (vertical, ground) [lux]
τ	Light transmittance of the glazing [%]

The overall illuminance (E_{i70}) is obtained as the sum of all the single cells results.

Thermal Module

This module calculates the energy consumption due to heating and cooling. It considers thermal transmission as well as radiation.

The energy flows are calculated as follows:

$Q_H = \sum (U_i A_i) \cdot DD_H$

$Q_C = \sum(U_i A_i) \cdot DD_C + SHG \cdot SHGF \cdot A_{tr}$

and finally:

$$Q_{tot} = Q_H + Q_C$$

where:

$Q_{H,C}$	kWh/m^{2}y	Energy required for heating / cooling
$U_{tr,op}$	W/m^{2}K	Coeff. for heat transmission (transp / opaque)
$A_{tr,op}$	m^2	Area (transparent / opaque)
$DD_{H,C}$	d°C	Degree days for heating and cooling
SHG	W/m^{2}y	Solar heat gain per one summer
SHGF	%	Portion passing through glazing
Q_{tot}	kWh/y/m^2	Total energy consumption

Architecture Module

This module ensures that the density of transparent cells increases toward the center of the façade, for the occupants to have a view to the exterior. In addition it also tends to increase the sizes of the cells toward the center of the façade.

The zone of preference for transparency is defined by a preference matrix, with as many elements as cells, whose values are higher where more transparency is needed.

Multiplying this preference matrix by the actual transparency distribution on the façade, and summing up the terms, provides the rating for architecture.

$$J_{arch} = \sum_{i=1}^{\#cells} DV(mat)_i \cdot Z_i$$

where:

DV(mat)i	is the degree of transparency of cell i (1, 2, or 3), and
Zi	is the desired zoning preference for that cell.

A similar rating function is also used to increase cell sizes towards the center.

Structure Module

The structural module assesses the physical efficiency of the design.

A static solution is then calculated and, based on the ratio stress/capacity in the members, an overall rating is returned. The module calculates the stresses in the members and returns to the optimizer a rating function, which "grades" the input geometric pattern from a structural point of view.

In analytical terms, the rating function has the following form:

$$Rf = 1 - \sum_{i=1}^{N} \left(1 - \frac{|\sigma|_{\max,i}}{f_y} \right)^2$$

where:

$\sigma_{max,I}$ is the maximum stress (absolute value) in member *i* [MPa]
N is the number of elements
f_y is the yield stress of steel [MPa]

Economy Module

The economy module calculates the cost of glass and cladding materials associated with the design vector. The module considers first material costs and second costs per piece:

$$C = \sum_{i=1}^{3} c_{A,i} \cdot A_i + \sum_{i=1}^{3} c_{n,i} \cdot n_i$$

where:

$c_{A,i}$ is the cost of material i per unit area
A_i is the area covered with material *i*
$c_{n,i}$ is the cost of installation of material *i* per cell
n_i is the number of cells with material *i*

This module provides a simple approximation of the manufacturing and installation costs.

9.2.4.3 Evaluation

Two different approaches were implemented within the different experiments to asses and evaluate the multi objectives within the objective function. Initially a *weighted sum* approach was used, and later a *utility functions* approach was adopted.

The weighted sum approach was chosen for combining the distinct quantities. Disparities certainly exist between the magnitudes of each quantity. However, foreseeing the problem, these quantities were normalized based on reference values. These reference values represent the achievable maximum in the optimization considering only one objective at a time.

The objective function *J* for the weighted sum approach has the following form (note that it was used for the first experiment, and therefore only contains terms related to thermal, architecture and lighting):

$$J = \frac{1}{3}\left(w_1 \cdot \frac{1}{\frac{Q_{tot}}{r_1}} + w_2 \cdot \frac{Z_{tot}}{r_2} + w_3 \cdot \frac{E_{i70}}{r_3} \right)$$

where:

r_1, r_2, r_3	are reference values used to scale down the output of the computational modules, and
w_1, w_2, w_3	are rating coefficients that “weight” the importance of each single objective function.

This objective function attempts to reflect some of the trade-offs between sufficient natural lighting in a room, the energy balance due to cooling/heating of the façade, and the architectural intent to have windows in view height.

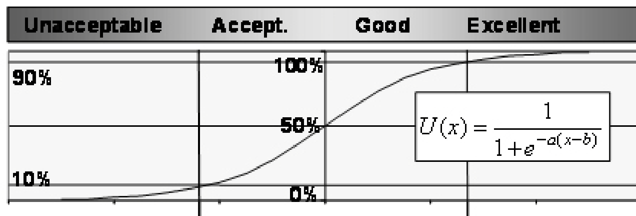
Later, in the attempt to develop a more rigorous objective function, a utility approach was adopted. Because of the multi-objective nature of the skin design, each of the modules will have its own single objective (total illuminance, energy required for heating and cooling, etc…). Each single objective J_i is then transformed by a utility function U_i, that assigns a low score to an undesirable value of J_i and a high score to a desirable one ($0 \le U_i \le 1$).

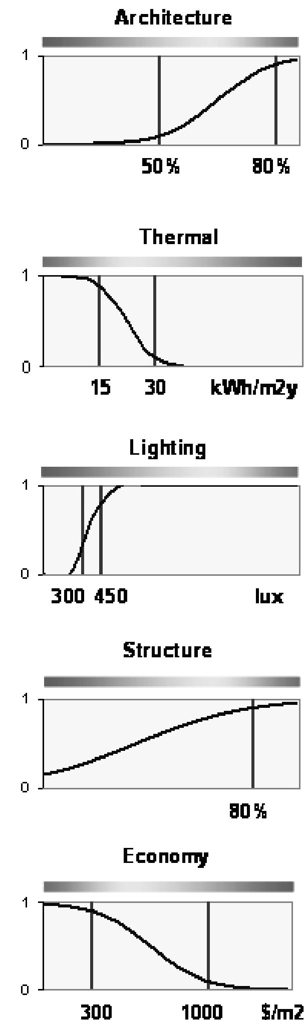
The utility functions are scaled in such a way that an excellent performance is rated more than 90%, while an unacceptable performance has a rating less than 10% (figure 9.26). The latter is

also considered as a minimum performance determining the feasibility of a design (Figure 9.26).

Figure 9.26:

Structure of the utility functions





Module	Optimum	Range	U = 0.1	U = 0.9	Units
Thermal	Min	[0; 70]	30	15	*kWh/yrm^2*
Architecture	Max	[0; 1.1]	0.5	0.85	-
Lighting	Max	[0; 2000]	300	450	*lux*
Structure	1	[0; 1.1]	0	0.85	-
Economy	Min	[0; 1500]	1000	300	-

The application of utility functions was believed to be necessary because for some objectives a “the more the better” or “the less the better” approach is not applicable. For instance, an illuminance between 300 and 450 lux for lighting is desirable, but to increase the illuminance does not improve the usability. In contrast, it causes problems of glare in areas close to the windows. An overall objective function U takes into account all the multi-objective particular utilities U_i. The U function has the following form:

$$U = \sum_{i=1}^{5} w_i \cdot U_i$$

Where w_1, w_2, w_3, w_4, w_5 are weighting coefficients that depend on the importance of each single objective function.

This overall utility function reflects the trade-off between sufficient natural lighting in a room, the energy balance due to cooling/heating of the façade, the intent to have windows in view height, the requirement for structural safety, and limited cost.

9.2.4.4 Optimization

Within the different experiments, two optimization approaches were used: a gradient based and a heuristic algorithm search. In Experiment#1, the results of these two different optimization techniques were assessed and compared. In the rest of the experiments, only the heuristic methods were implemented.

Gradient Based Optimization

The Gradient based algorithm selected for the optimization was Sequential Quadratic Programming (SQP). It is a widely used method in most engineering applications (like non-linear optimization), and it is considered to be a robust gradient-based algorithm. It is especially reliable because of its strong theoretical basis. The optimization was performed using ModelCenter 6.1.1™, from Phoenix Integration Software.

Heuristic Algorithm Optimization

The Heuristic method used in the experiments was a Genetic Algorithm (GA). GA's are heuristic algorithms that utilize processes analogous to natural selection to search for the best designs. They were chosen because they are ideally suited for design problems with discrete design variables. Because they do not require objective or constraint derivative information, they are able to effectively search non-linear and noisy design spaces. The optimization was performed using the Darwin Plug-In of Model Center 6.1.1™, from Phoenix Integration Software.

9.2.5 Integration

Model Center 6.1.1 was used to build the system architecture (figure 9.27), by integrating the modules, following the overall conception of the four design activities phases (synthesis, analysis, evaluation

and optimization).

The geometry module was built using CATIA™ of Dasault Systems. The lighting, thermal, architecture, and economy modules were developed within Microsoft Excel™ XP. ANSYS™ 10.0, from Ansys, Inc., was used to construct and execute the structural analysis.

To assess the results of the optimization, a few visualization tools were developed. For experiment #1, a material visualization tool was programmed in Excel. To display the evolution of the optimization results. For experiments #2 and #3 another tool was developed in order to visualize the geometry of the optimization solutions; the best solutions generated were also recorded. A common interface was developed, linking and showing the current design, its geometry and materials, and the time-history of the objective function, in one common interface (Figure 9.28).

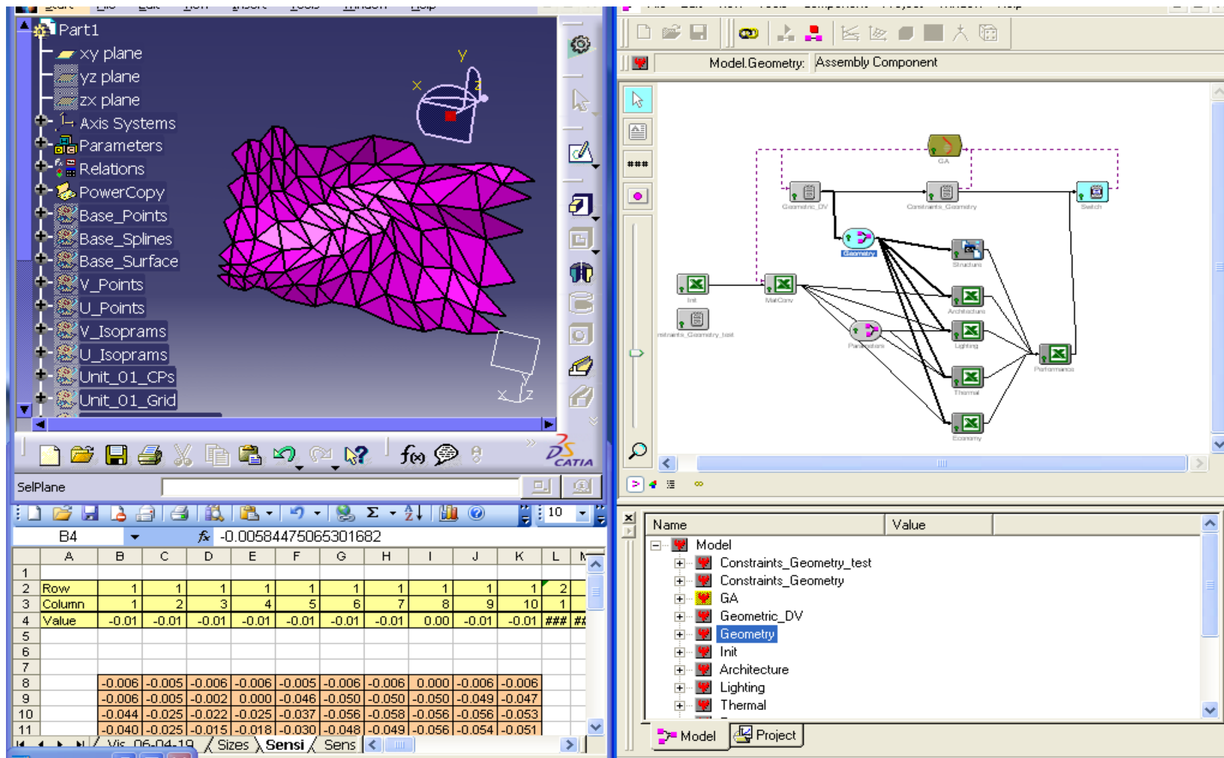
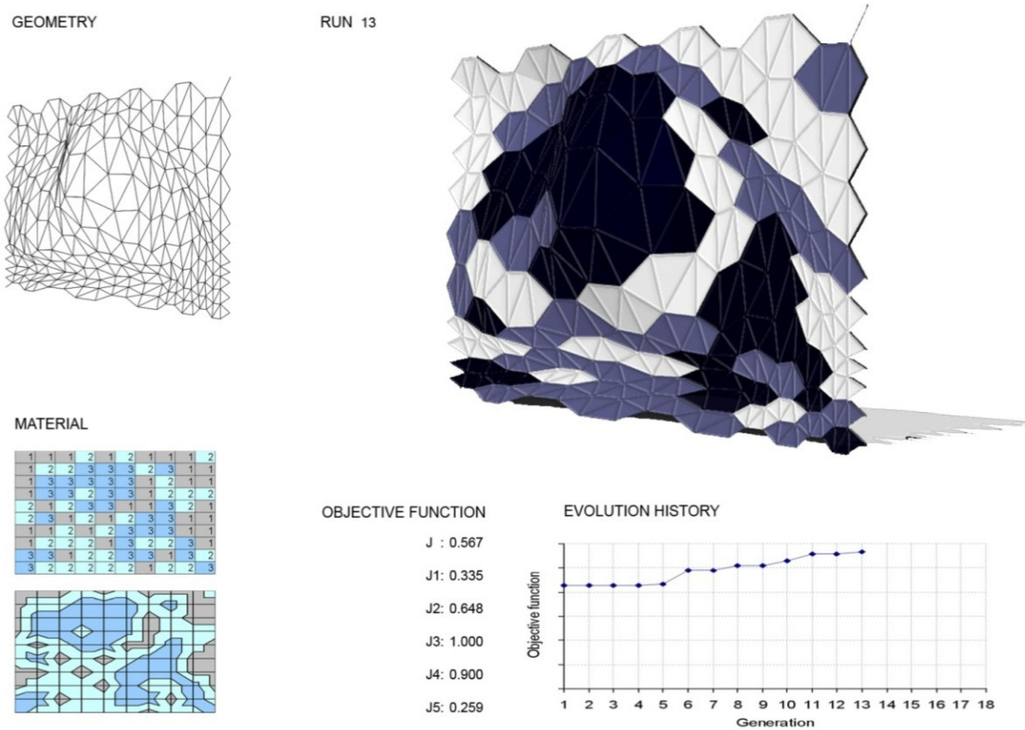


Figure 9.27:

Integration between different software

Figure 9.28:

Interface that plays back Evolution History



9.2.6 Exploration

9.2.6.1 Experiment 1

For the first experiment, the material variables were the only part of the design vector in the synthesis phase that were allowed to vary. Geometry was constant , i.e. the skin configuration was “frozen”. As a consequence, the design vector is only populated with material variables. A grid of 10x10 cells was selected, and the material variables were considered continuous as opposed to discrete to allow for the implementation of different optimization algorithms.

Within the Analysis phase only a subset of the analysis modules was selected for this experiment, namely, the thermal, the architecture, and the lighting modules. The problem presented in this experiment was solved using two algorithms: first, by using a gradient search

(SQP, Sequential Quadratic Programming), and later, by using a heuristic technique (Genetic Algorithms).

To asses any scaling concerns, a script was developed to evaluate the diagonal terms H_{ii} of the Hessian matrix. All H_{ii} terms had the order of magnitude of unity. This was due to the fact that all the design variables represent the same physical object, i.e. the degree of transparency of the cells, and therefore have the same boundaries (0 and 1). It naturally follows that the problem was intrinsically well scaled with respect to the design variables (in fact, scaling is necessary when the design variables can assume very different values). Thus, no scaling of the design variables was necessary.

For this experiment, two types of constraints are present: firstly, each design variable has a lower and an upper bound, respectively 0 and 1; as a consequence, there are 2 x 100 = 200 constraints of the first type (*transparency constraints*, [TC]). Each TC affects only the correspondent design variable, so there can be no disparity in sensitivity with respect to different variables.

Secondly, the constraint on the overall illuminance sets E_{i70} > 300 lux; this *illuminance constraint* [IC] affects all the design variables with different sensitivity, because not all the cells have the same influence on the overall illuminance due to their different positions on the façade and to their surface areas: the higher the location and the wider the surface, the higher the influence. Nonetheless, the ratio between the highest and the lowest of the sensitivity coefficients is not big enough to justify a scaling of the constraint.

With regards to parameter sensitivity, the objective function increased when the spatial component depth was increased, since the associated region surface area enlarges. In fact, within the thermal module the objective is expressed as the required energy per floor area. The other modules outputs do not demonstrate any change.

Increasing the height of the spatial component enlarges the skin area, and therefore results in a decrease of the objective function. In fact, the thermal objective decreases while lighting is unable to compensate.

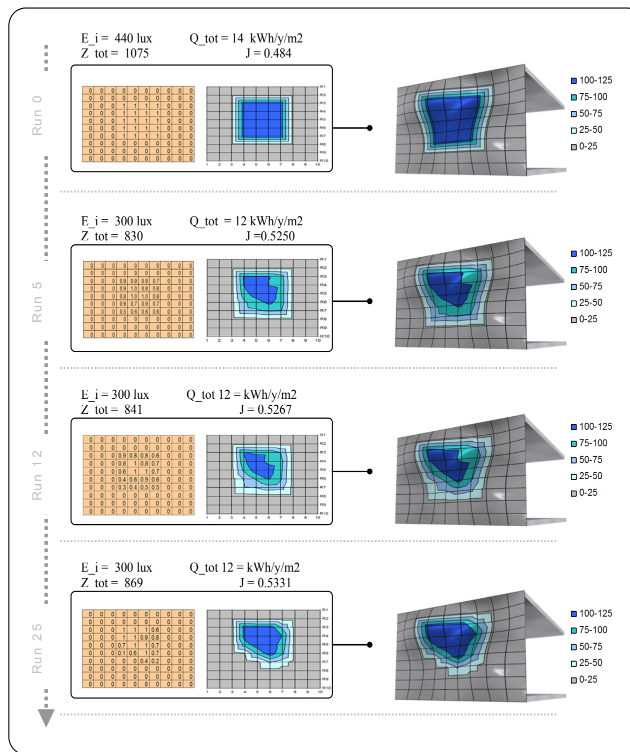
Increasing the light transmissions of the glass increases the lighting objective. However, since a higher thermal transmission of glass leads to a higher energy consumption, and therefore lowers the performance of the thermal objective, the overall objective function is lowered.

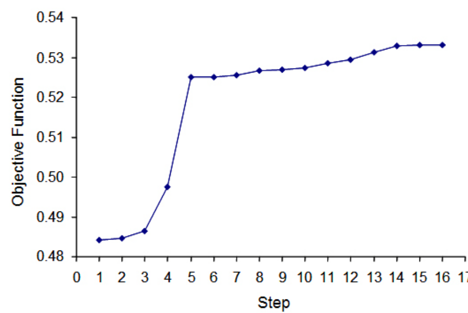
The illuminance was moved from 300 lux to a stricter value of 400 lux, and the optimization was rerun. The calculation with a higher value of the lighting constraint leads to a reduced performance in terms of the objective functions, the thermal aspect overriding the positive effects of the other modules.

For this experiment, the weighted sum approach was adopted to combine the distinct objectives. The overall objective function reflects the trade-off between sufficient natural lighting in a room, the energy balance due to cooling/heating of the façade, and the intent to have windows in view height. The request for day-lighting drives especially the cells at the top to be transparent to push light deeper in the room.

Figure 9.29:

Experiment#1 Evolution Of Design Using SQP





When the gradient search was used (SQP, Sequential Quadratic Programming), the objective function dramatically increased at the beginning of the analysis, and then, after finding the main illuminance constraint, slowly tuned to the potential optimum. As demonstrated in figure 10, a good improvement is obtained with respect to the initial point $\mathbf{x_0}$ (Figure 9.29). The solution corresponds to a value of the objective function of $J_{max} = 0.533$.

The cells that lie in the "magnification areas" of the lighting and architecture functions, i.e. respectively the cells on top and at the center of the façade (thermal, in fact, is not location-dependent), were those that massively turned transparent. This matches intuition since the cells at the center of the façade have a much greater weight due to the architecture module, and the cells on top of the façade "weight" more for the lighting module.

The active constraint is the minimum overall illuminance, plus a number of constraints for the transparency of cells, and precisely those of the cells that are *completely* transparent or *completely* opaque. (The design variables, at the end of the optimization, actually *assume* these limit values.) Given the objective function, the solution found seemed to match the physical assumptions of the model.

Later, when running the optimization using the Genetic Algorithms (GAs), which were terminated after 47 generations, the best recorded results were collected, and the optimal solution was found to corresponded to a value of the objective function of $J_{max} = 0.541$. (figure 9.30). The above result was obtained by fine-tuning the algorithm parameters, which are listed hereafter:

Population size:	30
Preserved designs :	13
Max No. of generations:	50
Mutation Probability:	0.05
Selection scheme:	Multiple Elitism Selection

This result is higher than the previous one found with a gradient-based search (Sequential Quadratic Programming).

In order to understand the reasons of this difference, we studied the physical implications of the result. In particular, after a first analysis of the design vector, it was clear that the number of completely transparent cells is higher and, given a cell, its degree of transparency is generally higher. Also, the "transparent zones" were not just confined to the center and to the upper portion of the façade, and the illuminance constraint is far from being active, its final value being 1441 lux.

In other terms, starting from the "optimum" found by the means of a gradient-based algorithm, the GA's provided evidence that the objective function could be "pushed" further up by adding more glass, i.e. by turning more cells transparent. It can be inferred that the solution found by the SQP algorithm is not the real optimum, since a better one exists. Alternatively, it was proven that the gradient-based algorithm found a local optimum.

9.2.6.2 Experiment 2

For this experiment, the nodes were allowed to move in the u and v directions on the skin surface. The materials of the cells were also allowed to change, but using discrete values for variables. The 10x10 cells grid used for experiment#1 was kept for experiment #2.

As a consequence, the design vector is populated with the coordinates of the 32 Control Points (2 coords per central CP, 1 coord per border CP) and of the material variables , for a total of 48 + 100 = 148 variables. Within the Analysis phase, the modules that were already part of the system architecture of experiment #1 were used – i.e. the thermal module, the architecture module, and the lighting module –; in addition, a further economy module was added, to minimize the cost of cladding materials.

For this experiment the utility function approach was used, in order to take into account multi-criteria optima.

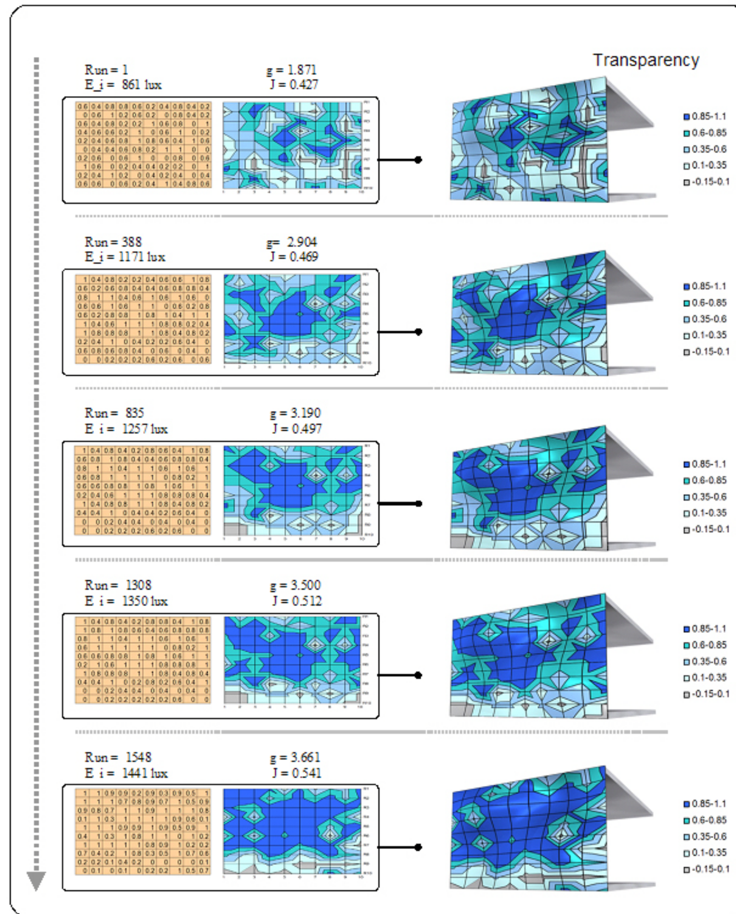
The problem presented in this experiment was solved using Genetic Algorithms. The Genetic Algorithms were terminated after the max number of generation (30) was reached, and the best recorded results were collected. The optimal solution that was found, corresponded to a value of the objective function of $J_{max} = 0.567$ (Figure 9.31).

The above result was obtained by fine-tuning the algorithm parameters, which are listed hereafter:
Population size: 25
Preserved designs : 13
Max No. of generations: 30
Mutation Probability: 0.06
Selection scheme : Multiple Elitism Selection

Note this value of the objective function cannot be compared to values of the values obtain in experiment #1, due the fact that we used a different objective function. The optimal solution that was found, corresponded to a value of the Objective.

Figure 9.30:

Experiment#1, Evolution of Design using GA's



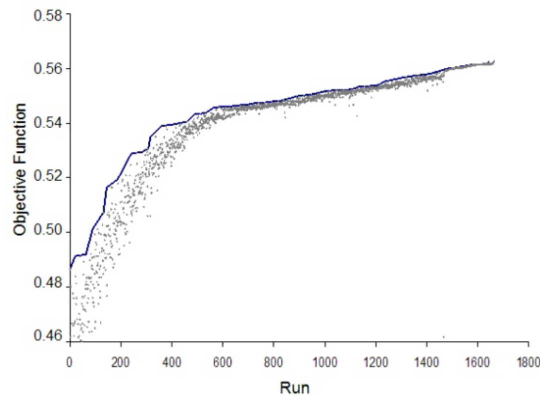
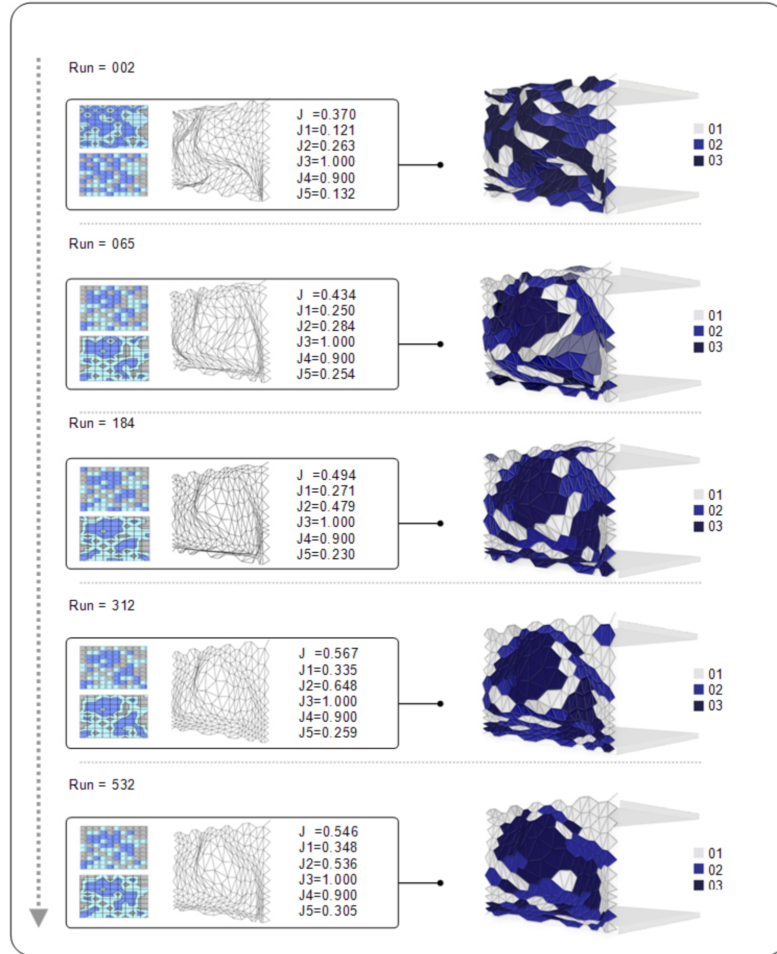
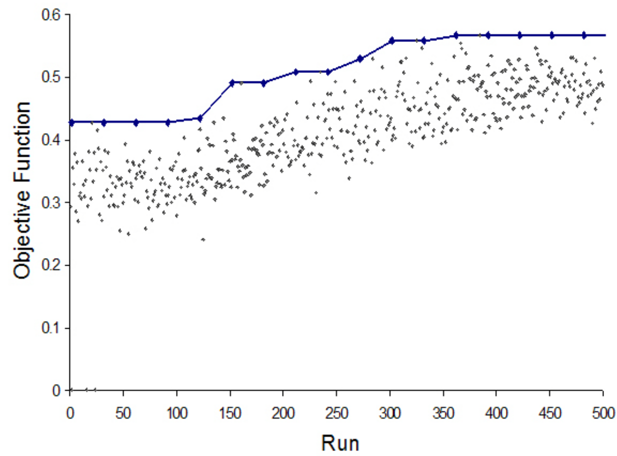


Figure 9.31:

Experiment#2,
Evolution of Design





The solution shows a particular geometrical and material pattern. The cells correctly enlarge at the center of the skin, and they shrink as the distance from the center increases. In addition, in the center of the façade a complete region turned to glass, a consequence of the architectural module action.

Away from the center, the density of semi-transparent and opaque material grows higher, a sign that a trade-off was reached between lighting and architectural requirements on one side, and thermal requirements on the other.

To compensate a higher density of transparent cells in the center of the skin, which allow for a more abundant light intake, the cells close to the border turn opaque to limit the heat loss. This configuration slowly becomes evident as the number of generations grows. The central pattern of transparent cells develops from an early embryo, to fully occupy the whole central region of the skin.

In the distribution of the non-transparent cells, the partition between the semi-transparent and the opaque ones does not seem to follow a particular scheme.

It was not possible to understand if a distinguishable pattern might have been achieved after more generations. In any case, the slow progression of the overall objective function toward the end of the process suggests that any further variation in the design vector cannot generate a substantial change in the objective, from which it may be inferred that the sensitivity of the objective with respect to that partition is not very significant. However, it is important to note that, like for all heuristic techniques, the convergence of the GA's cannot be mathematically proven.

9.2.6.3 Experiment 3

For this final experiment, the nodes of the skin were allowed to move in the three directions, u, v, and w. Like the previous experiments, the materials of the cells were also allowed to change, using three discrete values for variables.

Note that, due to the intense computational burden of the CAD systems and the FEA Analysis, instead of the 10x10 cells grid of experiment#1 and experiment #2, a 6x6 grid was used.

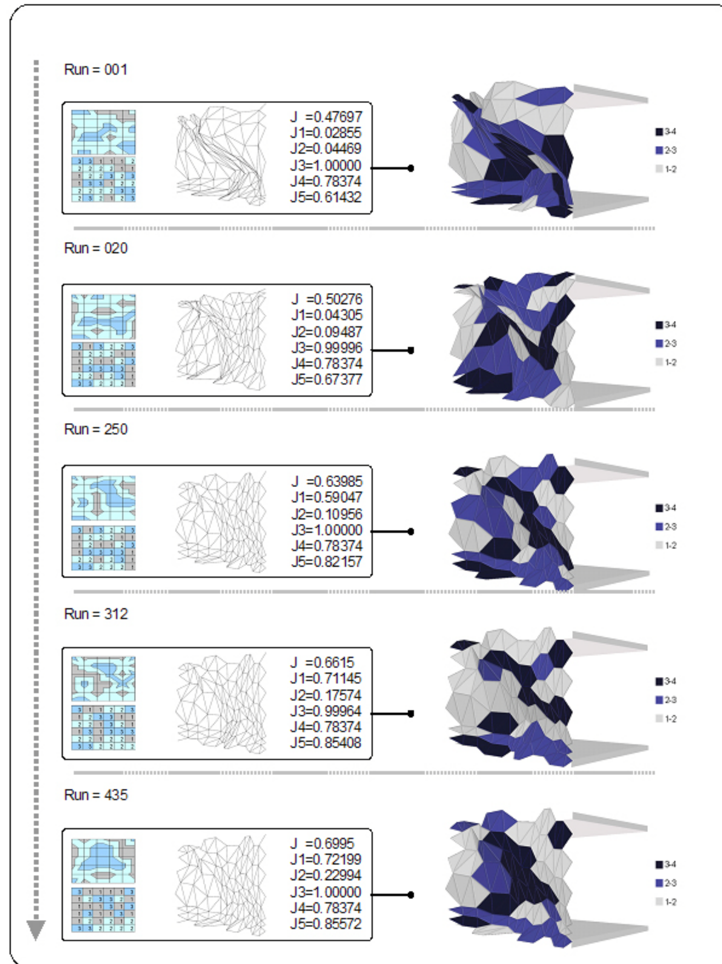
As a consequence, the design vector is populated with the coordinates of the 12 control points (4 in the middle, plus 8 on the borders, 2 coords per central CP, 1 coord per border CP) and of the

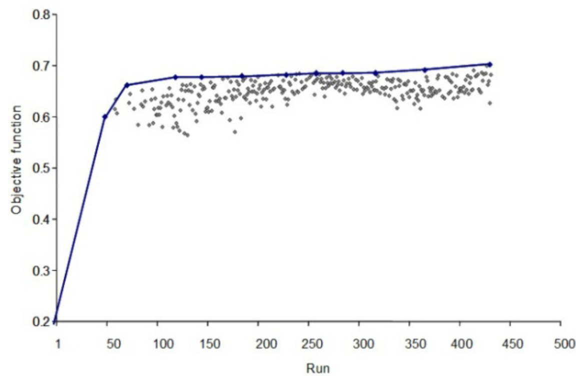
material variables, for a total of 16 + 36 = 52 variables.

Within the analysis phase, the modules that were already part of the System architecture of Experiment #1 and #2 were used; in addition, a further structural module was added, to assess the efficiency of the grid from a structural viewpoint.

Figure 9.32:

Experiment#3,
Evolution of Design





Much like the previous experiment, the utility function approach was used to combine the single-discipline objectives, as described earlier.

The problem presented in this experiment was solved using Genetic Algorithms. The GA's were terminated after 35 generations, and the best recorded results were collected.

The optimal solution that was found, corresponded to a value of the objective function of $J_{max} = 0.699$. (figure 9.32).

The above result was obtained by fine-tuning the algorithm parameters, which are listed hereafter:

Population size : 40
Preserved designs : 13
Max No. of generations: 40
Mutation Probability : 0.06
Selection scheme : Multiple Elitism Selection

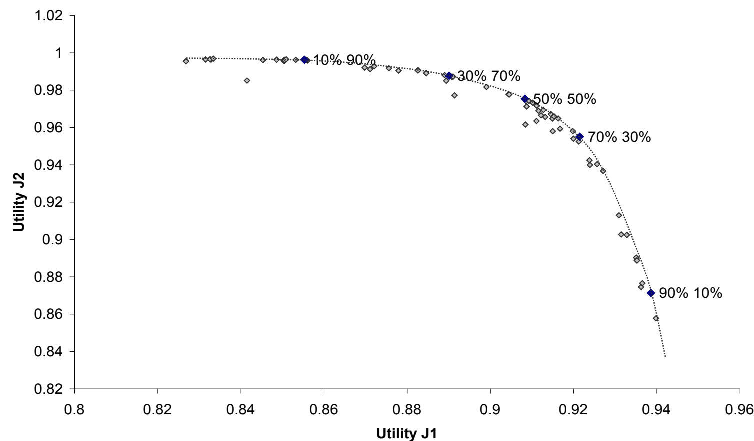
Note this value of the objective function cannot be compared to the values obtained in the previous experiments due to the fact that a different design vector was used.

Similarly to the previous experiments, the evolution of the design along the generations of the GA's can be mapped by analyzing previous solutions. Five evenly spaced in "time" solutions were chosen.

The graphical rendering of the solutions proved to be a helpful tool to visualize the formation of geometric trends and patterns in materials distribution (figure 9.34).

Figure 9.33:

A Pareto front based on the thermal and lighting objectives

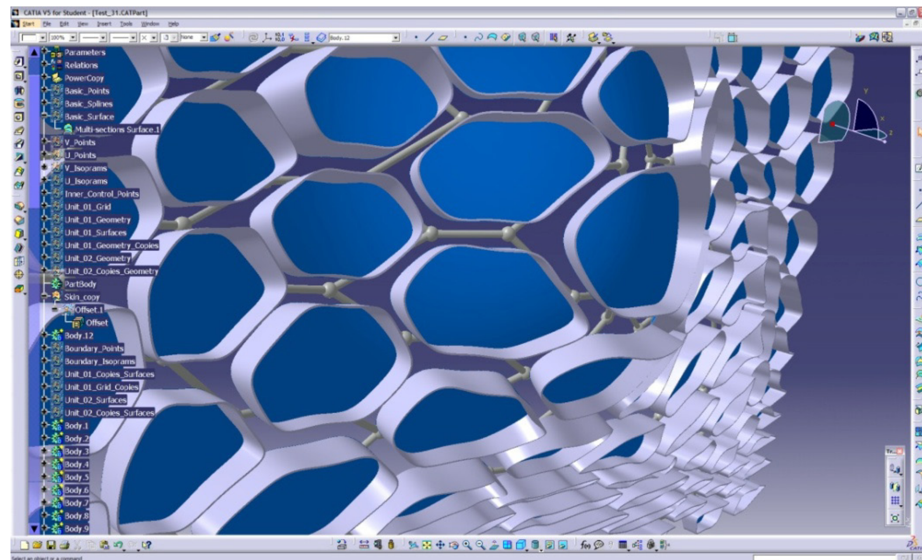


Consistently with the assumptions underlying the form and the function of the structural module, it can be observed that the design shows a constant tension toward geometric regularity: after the first runs, where the skin is characterized by very different cell sizes – and, as a consequence, by different beam length – the design of the skin becomes more and more organized in a regular disposition of cells with more evenly distributed surfaces and more similar shapes.

Nonetheless, it still holds true that the central cells are wider than the ones located near the borders, to reach a compromise with the architectural objective.

Figure 9.34:

A graphical rendering of a solution



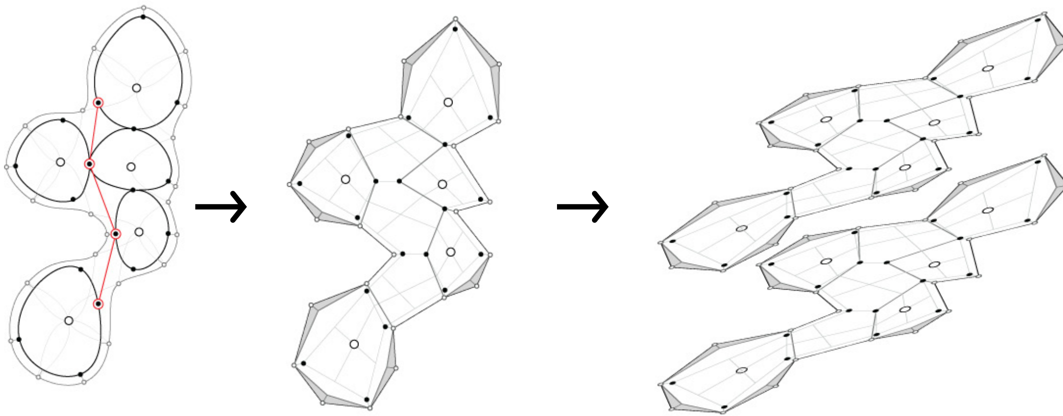
9.3 Experiment 2 | Level 1

9.3.1 Design Concept

Figure 9.35:

Diagrams explaining the changes in the Design Concept

The design concept of this experiment builds on the concept developed in Experiment One. It uses the same logic of allocating the *spatial components* within the rectangular site but within a 3D grid of cells, instead of the 2D grid. In addition the wrapper skin for the *spatial components* is constructed of straight lines as opposed to the spline used in experiment One. Within the new 3d grid, the spatial components are allowed to relocate and deform to satisfy multiple performance and objective requirements (figure 9.35).



9.3.2 Decomposition

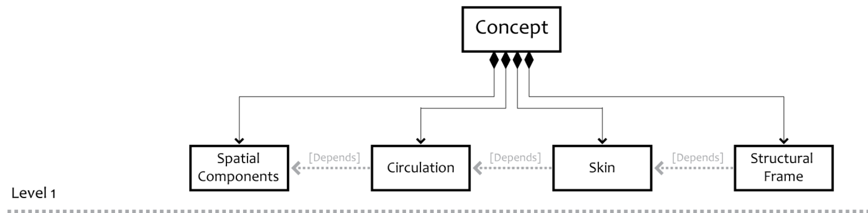
9.3.2.1 Component Decomposition

Similar to experiment one, the design concept will be decomposed initially into four main components. These will be the spatial components, the floors, the skin and the structural framing. There is a strong dependency between all four components in which changes in one component can affect the rest of the components (figure 9.36).

This dependency will have to be taken into account in the aspect decomposition. The synthesis modules will have to generate these different components.

Figure 9.36:

Component decomposition



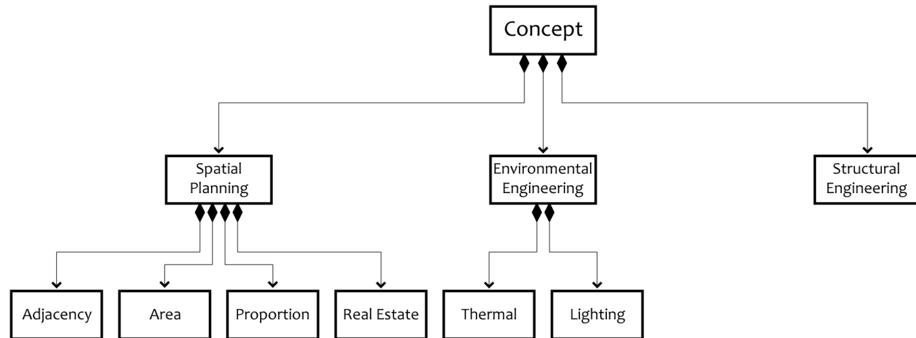
9.3.2.2 Aspect Decomposition

Based on the dependencies that exist between the different components in the initial level, the aspects of interest in all four components have to be identified simultaneously. Like experiment one, level one aspects will be decomposed into spatial planning, environmental, and structural aspects.

Similar to Experiment one, the spatial planning will be further decomposed into several lower aspects that include adjacency, area, proportion, and real-estate. The environmental aspect will be decomposed into two lower aspects that include thermal and lighting aspects. Structure will not be decomposed further (figure 9.37).

Figure 9.37:

Aspect decomposition

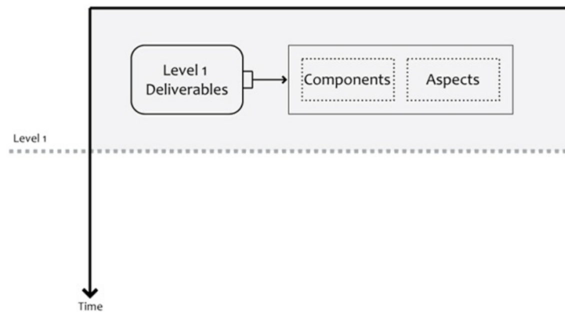


9.2.2.3 Development Decomposition

Within our current experiment, the deliverables of the MDDS level one will include a configuration of the spatial components, the floors, the skin and structural frame. These configurations will have to be assessed for adjacency, area, proportion, real-estate, thermal, and lighting and structure (figure 9.38).

Figure 9.38:

Development decomposition



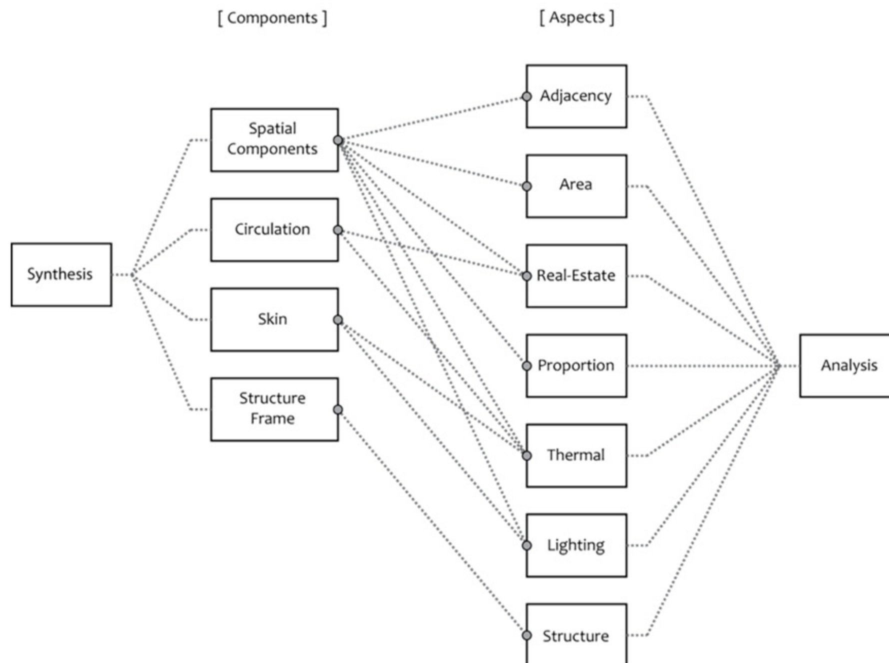
9.3.2.4 Activity Decomposition

Figure 9.39 shows the mapping between components and aspects. This identical to experiment one. However, unlike experiment one, in this experiment all aspects will be included.

This will produce only one design cycle for level one. This cycle will attempt to solve the spatial planning aspects, the environmental aspects as well as the structural aspects.

Figure 9.39:

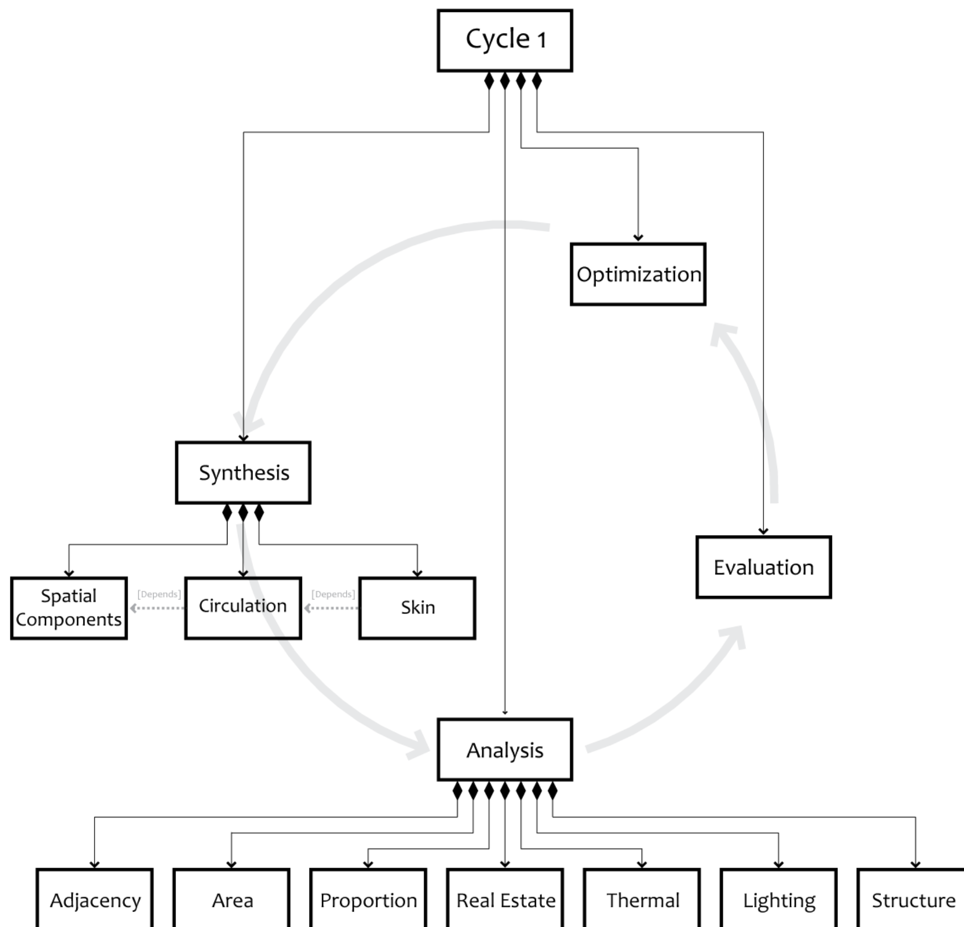
Component and aspect decomposition mapping



This cycle will be further decomposed into design activity modules. These design activity modules will include synthesis, analysis, evaluation and optimization modules. The Synthesis modules given a design vector will generate the spatial components, floor, skin and structural members. The analysis modules will analyze for adjacency, area, proportions, real-estate, thermal, lighting and structural behaviors. The evaluation modules will aggregate the different behaviors of the different analysis modules into a general performance quantity. Finally, given the outputs of the evaluation modules, the optimization modules will search the design space and specify a new design vector (figure9.40).

Figure 9.40:

Design cycle one and its design activities



9.3.3 Formulation

What is interesting in this experiment is that the same MDDS architecture for the first cycle in level one which was used in

experiment one, is reused in this experiment with minimum modifications. These modifications are demonstrated in figure 9.41. Two new synthesis modules were added to the synthesis cluster. In addition a new structural module was added to the analysis cluster. This ability to reuse architectures and modules is one of the strengths of the modular approach used in the MDDS.

As stated in the decomposition section, the synthesis modules given a design vector should generate the spatial components, floor, skin and structural members. This will be achieved by an assembly of three synthesis modules. The first will generate the general configuration and geometric information needed by most of the analysis modules. The second module will generate information relevant to the environmental aspects that were not generated by the first module. The third module will generate the relevant structural information that was not included in the first module. These three modules will be connected together and will function as a synthesis assembly.

This synthesis assembly will receives the design variables from the design vector's module in the optimization cluster and will output a design solution (phenotype) to be analyzed by the analysis cluster.

Within the analysis cluster, each of the seven analysis modules receives from the synthesis assembly the relevant information needed for its analysis. Each module should then provide a measurement of the design performance for its associated aspect.

In the experiment two modes of evaluation will be implemented. The first is a scalarization method which will be similar to the evaluation module implemented in experiment #1. This module will control the flow of data and if no constraints are violated, the performance and behavior measurements generated by the different analysis modules will be aggregated into an objective function that will be sent to optimization. The other mode of evaluation will be based on a Pareto filtering approach where the solution will be decided on after search. However this mode will still use the same control structures that filter out the solutions with high degrees of constraint violation. Again, similar to experiment #1, a GA was chosen as the optimization algorithm.

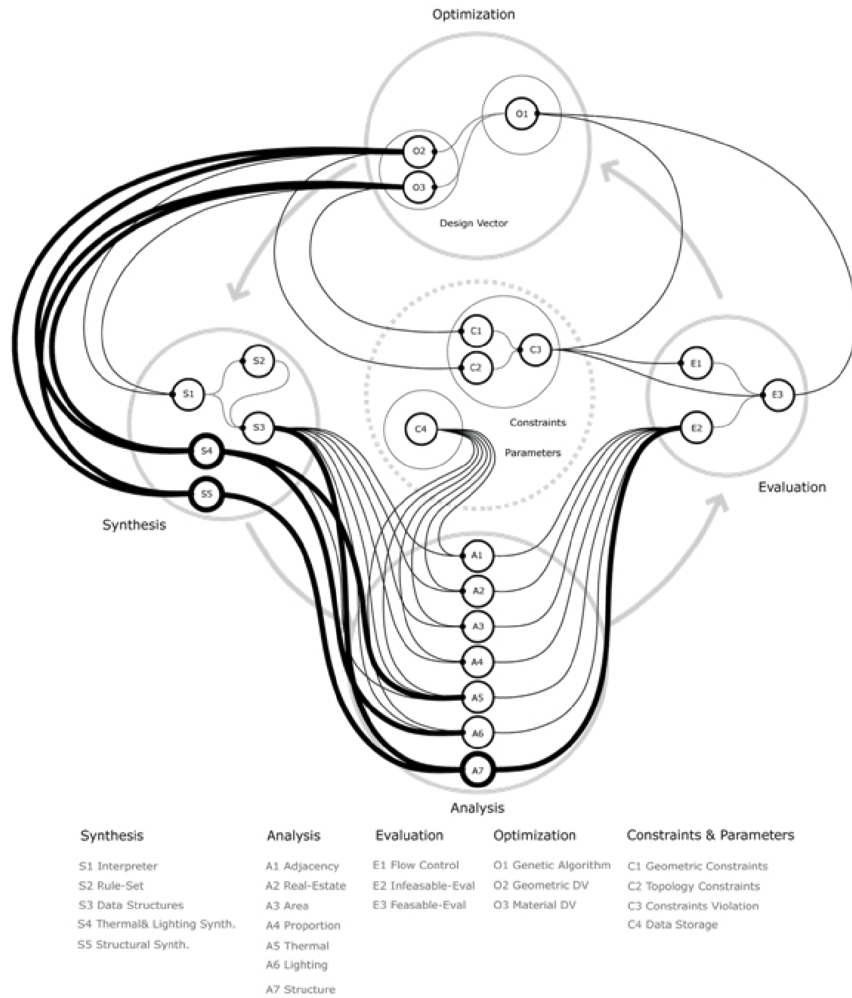


Figure 9.41:

The MDDS cycle on level one, showing extra modules and connections added

9.3.4 Modeling

9.3.4.1 Synthesis

The synthesis assembly in this experiment will build on the previous experiment modules, as well as implement extra modules to handle a more complex three-dimensional geometry.

The current synthesis assembly will consist of one sub-assembly and two modules. The sub-assembly will be a configuration assembly. This will be a modification of the synthesis assembly of experiment one and will generate the new crystallized three-dimensional geometries. The first of the modules will be a structural synthesis module which will generate the structural members for the configuration produced by the configuration assembly. The other

module will be a lighting and thermal synthesis module which will generate the shading patterns for the configuration produced by the configuration assembly.

Configuration Synthesis Assembly

This synthesis assembly will generate a configuration of spatial components. This assembly is more complex than the one developed in experiment one. Unlike the one floor design problem in experiment #1, this experiment includes two floors. A horizontal circulation spine is also added. In addition a vertical circulation component (stairs) is added. Therefore, in addition to the three main data structures; Cells, Spatial Components, and Skin, a circulation data structure is added.

Due to the 3D configuration, the shape grammar implemented in this experiment is more complex than the one used in the first experiment. The first set of design rules deals with the allocation of Spatial Components. The second set of design rules deals with the deformation of the Spatial Components. The third set is a parametric rule set that generates the skin directly. Each rule in the third rule set is applied based on the location of the component and the state of the surrounding cells. Each rule in this set also generates the skin *Regions* and each region in turn is broken down into segments. These regions and segments will be used by several analysis modules.

Structure Synthesis Module

The structure of any configuration will be divided into two structural systems: a main structural system that supports the main loads of the building, and an exoskeleton system that supports the skin cladding. Both systems are generated automatically from a given spatial component configuration using several rule sets. In addition, this module identifies components that are not supported by lower components. Supports are then added to each component in the cantilever list using context sensitive rules. Given the occupation state of the surrounding cells the rule set adds a certain type of support to the main structural system.

This module then outputs to the analysis modules all the information related to the geometry, type and number of members as well as information pertaining to the cantilevering spatial components.

Lighting and thermal Synthesis Module

Due to the existence of cantilevers, spatial components in the lower

level can be affected by shading. The lighting and thermal synthesis assembly creates a shading pattern of a given configuration that will be used later by both the lighting and thermal analysis modules.

The synthesis algorithm that generates the shading pattern is based on the underlying cell grid. Each internal cell has three internal sides and one exterior side (N, S, and E or W). While each end cell has two internal and two external sides. The algorithm tests each cell and its surrounding cells for occupancy. If a neighboring cell is occupied then the corresponding side in the cell being tested is disregarded since that implies that there is no cantilever from that direction. However, for the sides that are not blocked by any components on the lower level, another test is applied to determine if there are cantilevers from that side. A correction penalty is then added if a cantilever exists. This penalty factor depends on the area and depth of the cantilevering spatial component.

Design Vector

Similar to experiment one, the design vector that provides the inputs to the synthesis modules is divided into two types of variables, namely topological and geometrical. However, the size of this design vector is significantly larger than the one implemented in the first experiment. All three synthesis modules were implemented in the CATIA VBA environment.

To summarize, the inputs to the synthesis modules are:
a- Location of the Spatial Components which is represented by the topological variables
b- Location of the Control Points in the system which is represented by the geometrical variables.

And the outputs are:
a- Area of each Spatial Component
b- Perimeter of each Spatial Component
c- Length of each Spatial Component Region
d- Length of each Segment composing a Region
e- Orientation of each Segment composing a Region
f- Total Area enclosed by Skin.
g- Structural members attributes (number of member, type, ..etc)
e- Shading patter of a configuration

9.3.4.2 Analysis

Like experiment one, the design concept will be broken down into multiple single-disciplinary analysis modules in order to evaluate how

well the design performs from the point of view of each discipline separately.

Several modules from experiment one will be reused after some modification to accommodate the changes in the number of spatial components as well as changes in geometry. These modules include: an Adjacency module, an Area module, a Real-Estate module, a Proportion module, a Thermal module, and a Lighting module. Furthermore, a new module will be added for structural analysis.

Once again this demonstrates the strength and adaptability inherent in the modular approach used in MDDS.

Since we are working at the conceptual stage, the analysis models developed for the different discipline modules will be based on heuristics or simplified representations to test the feasibility of design solutions. The modules are implemented directly in VB Scripts or in Excel using VBA Scripts.

For reasons of brevity the focus in this section will be on the structural module which is the only new module not discussed previously in experiment one.

Structural Module

The Structural Module evaluates the performance of the structure within the configuration using two criteria: Regularity (/Utility) and Cost.

The Structural Module receives information from the synthesis modules such as the coordinates of the CP's and the location and type of cantilevering cells among others, and outputs an overall evaluation of the structural performance, J_{Stru},.

Regularity

The algorithm provides an estimate of the structure regularity along the longitudinal axis of the building configuration.

Regularity is intended to be optimum if all the projections of the distances between consecutive CP's along the Y-axis (longitudinal axis) have the same length. Any deviation from ideality is captured by the following formula:

$$J_{R,1} = \sum_{i=1}^{8} \left| CP_Y_{i+1} - CP_Y_i \right|$$

Such calculation, shown for the first columns ($J_{R,1}$), is performed for all the three columns of CP's in the longitudinal direction, and results are summed to obtain an overall value J_{Regu}. Note that such sum has a lower bound of zero, corresponding to optimality.

Cost

Costing is partitioned in: cost of slab and beams, cost of columns, cost of bracings and cost of connections.

Cost of slab and beams

A simple, heuristic method of calculating the cost of slab and beams is based on the geometry of each spatial component. Essentially, the cost for each bay (spatial component) is assumed to be proportional to the longer span and to the square of the shorter span (i.e. the direction in which the slab spans):

$$C_s = \sum_i C_{s,i} = \sum_i \beta \cdot {b_{s,i}}^2 \cdot b_{l,i} \ ,$$

Where:

β is a fixed coefficient that depends on material properties, and is then fixed once such design choice is made. As such, a relative comparison of the performances of several designs can be performed based solely on the geometry.

$b_{s,i}$ is the shorter dimension of the spatial component considered

$b_{l,i}$ is the longer dimension of said component

Cost of columns

The cost of the columns is calculated based on the consideration that the cost is proportional to the section, which in turn is proportional to the tributary area and to the loading. As such, columns are grouped in five categories, according to the usage of the surface above:

Type I: Lower floor, no cells above
Type II: Lower floor, cells above
Type III: Lower floor, cantilever A above
Type IV: Lower floor, cantilever B above
Type V: Upper floor

Each category has a characteristic value of the cost, which is multiplied by the area of the surface:

$$C_c = \sum_{k=1}^{5} P_k A_k \ ,$$

where:

k is an index that relates to the aforementioned types of columns

P_k is the cost of the columns of type k per tributary unit area

A_k is the tributary area for type-k columns

Cost of braces

This parameter is kept constant, as the dimensions and section sizes of the bracing elements are not expected to vary substantially with the design.

Cost of connections

The cost of connections is held proportional to their number. The number of connections in a spatial component depends on its nature. Depending on whether the component is non-cantilevering, is a type-A cantilever or a type-B cantilever, the number of connections varies. As such, the total cost of connections results from the summation:

$$C_x = s(c_1 n_1 + c_2 n_2 + c_3 n_3)\,,$$

where:

s is the cost of one connection

c1, c2, c3 are the number of connections per type of component: non-cantilevering, type-A cantilever and a type-B cantilever

n1, n2, n3 are the numbers of components of type: non-cantilevering, type-A cantilever and a type-B cantilever, respectively

Overall cost

The overall cost is obtained by weighed summation of the above components, and needs to be minimised:

$$J_{\cos t} = k_1 C_s + k_2 C_c + k_3 C_b + k_4 C_x$$

Where:

k1, k2, k3, k4 are weighing coefficients that include all proportionality coefficients which were assumed as constant in the above singular costs

Combination of results

As J_{Regu} and J_{Cost} are not comparable dimensionally, weighing coefficients are used to derive the overall $J_{stru,o}$:

$$J_{stru,0} = a_1 J_{regu} + a_2 J_{\cos t},$$

where a_1 and a_2 are determined experimentally and kept constant along the optimization process.

Note that $J_{stru,o}$ which needs to be minimized, is finally filtered through the utility function *F* in order to obtain the definitive J_{stru}, which needs to be maximized:

$$J_{stru} = F(J_{stru,0})$$

F is monotonic, decreasing, always positive and its range coincides with (0, 1):

$$F = \exp[-\exp(\alpha_1 - \alpha_2 \cdot J_{stru,0})],$$

Where α_1 and α_2, too, are empirical coefficient that are kept constant throughout the experiment.

9.3.4.3 Evaluation

AS stated earlier in the formulation section, two methods of evaluation are implemented in this experiment. The First is a pre-search evaluation using a scalarization method and the second is a post optimization method that will use a Pareto filtering approach.

Both approaches will implement the *flow control module* that evaluates if the design vector violates the constraint modules. This module like in experiment one, acts as a switch directing the data flow to either of the other two evaluation modules. The other two modules are the *feasible design* and *infeasible design* modules. The flow control module triggers the infeasible design evaluation module if the constraints are severely violated. If the constraints are not violated the feasible design evaluation module is triggered.

If the design vector is infeasible the flow control module would bypass the synthesis and analysis modules saving extensive computational time. The infeasible design module simply signals the violation to the optimization modules and ranks the design solution in proportion to the number of violated constraints. The feasible design evaluation module on the other hand triggers the synthesis and analysis modules.

Within the first approach of evaluation where the scalarization

approach was used, The three evaluation modules implemented in experiment one were reused with little modification.

All the ratings (J_{area}, J_{circ}, etc.) of the disciplinary performances that originate from the analysis modules converge into the feasible design evaluation module, where they are aggregated to generate an overall evaluation of the design, according to the standard *scalarization* approach.

The final multi-disciplinary performance J is the weighted average of the normalized output from the various modules:

$$J = \frac{\sum_{m=1}^{M} w_m X_m}{\sum_{m=1}^{M} w_m}$$

where

$M = 7$ is the total number of analysis modules;
X_m is the normalized output from the mth analysis module, and w_m denotes the corresponding weight. In particular, X_m is obtained by normalizing the actual output of the mth module J_m according to:

$$X_m = \frac{J_m - J_{m,\min}}{J_{m,\max} - J_{m,\min}}$$

The second evaluation approach, the Pareto filtering, will be discussed in the exploration section after optimization.

9.3.4.4 Optimization

The optimization modules consist of two modules. The first contains the optimization algorithm, and the second is a design vector module which converts the outputs of the GA into data which the synthesis modules can understand.

Design Vectors Modules

As discussed earlier, two major categories of design variables have been considered in our experiment and are implemented in two different modules: the *topological* variables module and the *geometric* variables module.

The *topological variables module* defines the cell location of each Spatial Component. This module guarantees that no two Spatial Components are allocated in the same cell. For this experiment we

have eleven spatial components and eighteen potential cell locations.

The *geometric variables module* defines the ranges of the control points locations. In order to avoid any excessive distortion of the grid, each Control Point is forced to lie within a specific region.

Genetic Algorithm Module

Due to the nature of the design space, a Genetic Algorithm (GA) was implemented. The GA's genotype includes instructions for the synthesis assemblies to create a phenotype. In our experiment these instructions are of two types: *topological* instructions for allocating Spatial Components, and *geometric* instructions for modifying Control Points that affect the Cells. All the genetic transformations including crossover and mutation happen at the genotype level.

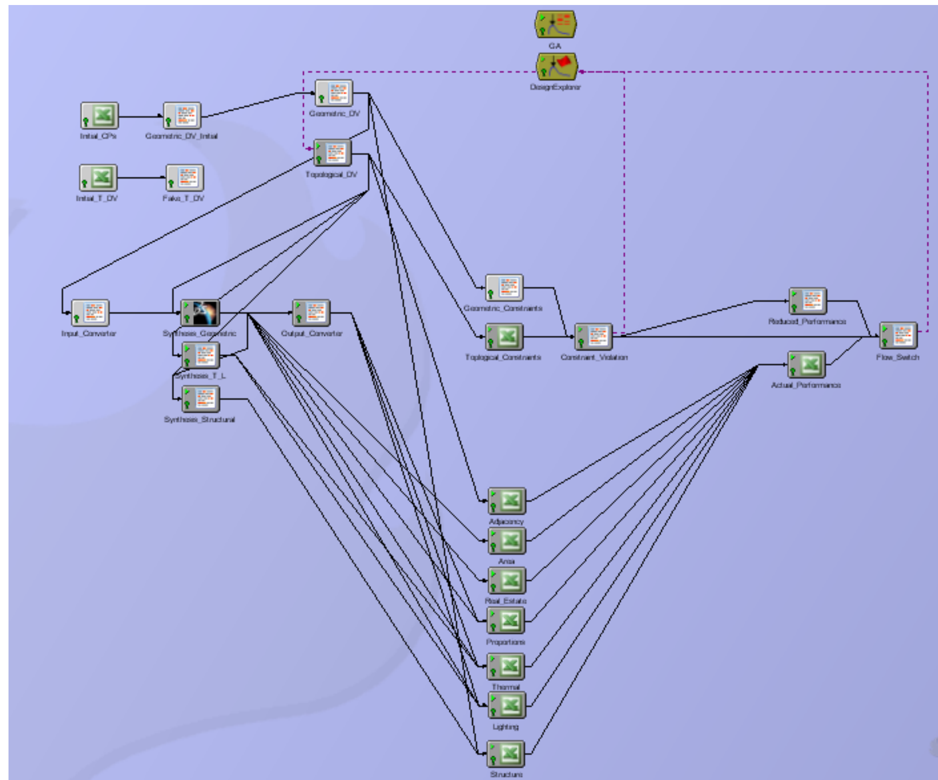
Constraints are implemented through the use of penalty functions. If a solution does not comply with the constraints in the system a penalty is added to the fitness of the design solution according to the degree of violation.

Due to the existence of multi-objectives the aim is not to produce a global optimum solution, but rather to direct the evolutionary process to produce populations of good solutions. These solutions would be used to study the tradeoffs between the different objectives. The GA's parameters used in the experiment were:

Population Size: 20
Maximum Generations: 500
Selection Scheme: Multiple elitist
Preserved Designs: 10
Operator Probabilities
  Discrete Variable Crossover: 1.0
  Discrete Variable Mutation: 0.15
Constraint Tolerance
  Maximum Constraint Margin: 0.05
  Percent Penalty: 0.5
Number of Top Designs Stored: 12
Random Number Seed: 3132

Figure 9.42:

MDDS Module Integration



Integration & Exploration

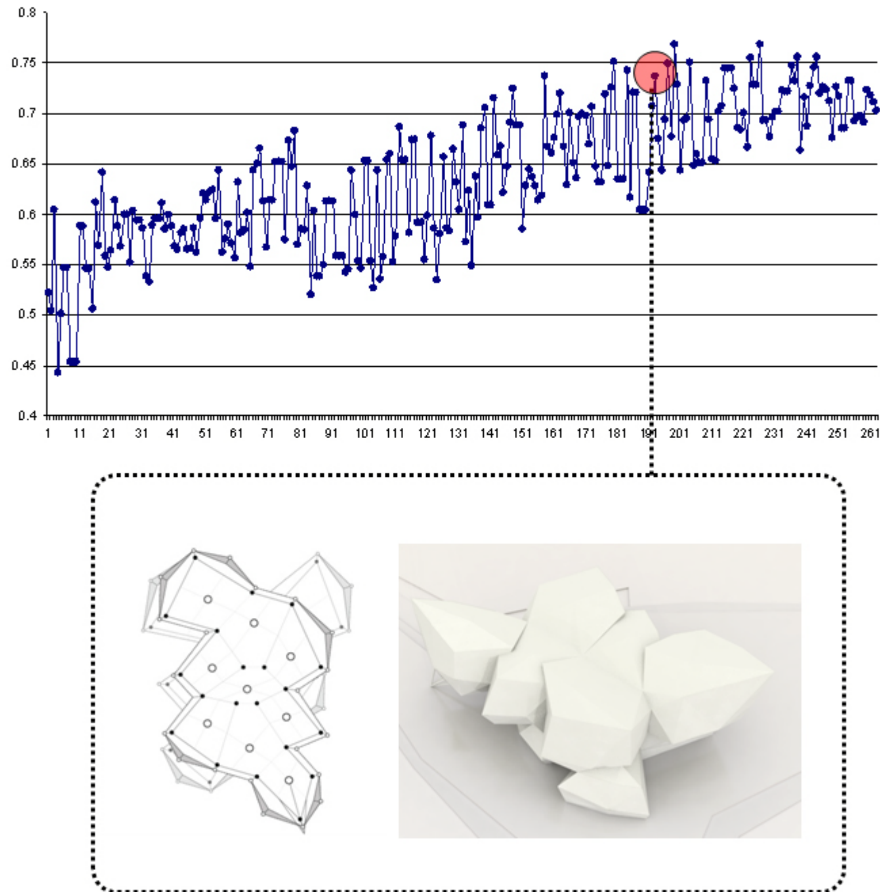
When all the modules discussed earlier have been built and their validity verified, the data flow model of the design system is implemented and the modules are integrated (figure 9.42). For the integration of the different modules Model Center from Phoenix Integration was used.

Optimization runs were started from initial seeded designs. As stated earlier two modes of evaluation were implemented. In the first mode a scalarization was attempted. The MDDS demonstrated interesting results (9.43).

The plotting of the search progression of the GA shows improvement to the overall performance of the design solutions beyond our initial seeded solutions. Similar to experiment one, design solutions in the final populations tended to be more compact in their shape.

Figure 9.43:

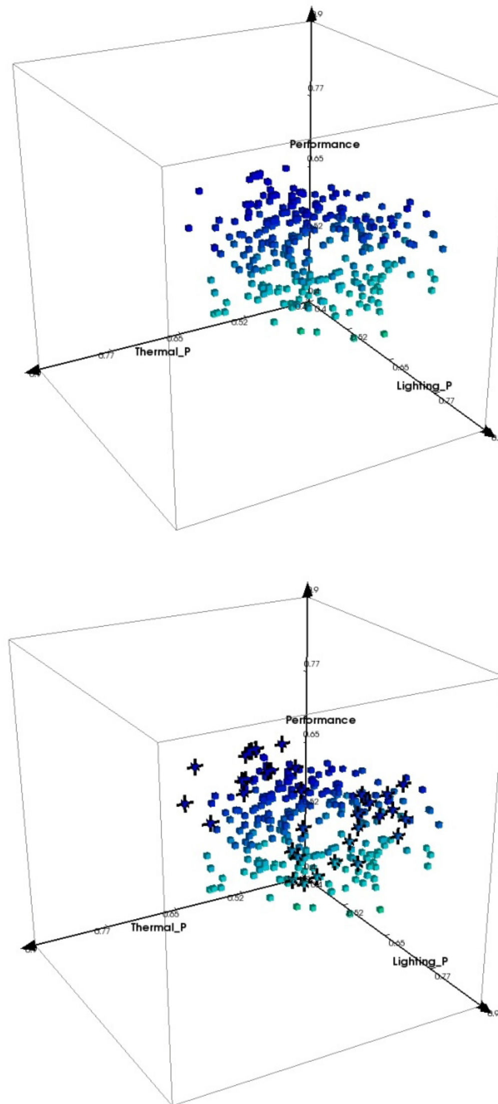
The MDDS evolution of solutions.



In the second mode of evaluation, a Pareto filtering approach was implemented. To assess better the trade-offs between the different objectives we needed to identify the non dominated solutions. This is because it focuses attention on a smaller set of solutions that are worth considering. These tradeoffs help select the final design.

Figure 9.44:

Pareto front of non dominated solutions



Despite the fact that this investigation is based on the results of a standard GA, the resulting feasible solutions were re-plotted in a 3-dimensional space in order to provide the design team some insight in the tradeoffs possible. These visualizations are useful because they show what needs to be given up in one objective to obtain an improvement in another.

The design team was interested in studying the tradeoffs between lighting and thermal objectives compared to the overall performance of the solutions.

Figure 9.44 shows two plots. The first is a 3D plot of solutions. The

graph axes represent the thermal and lighting objectives as well as the total objective which combines the rest of the design objectives. The second plot demonstrates the non-dominated solutions in the cloud of feasible solutions. These non-dominated solutions are obtained by a simple sorting algorithm and can be considered as a rough estimation of the Pareto-front.

As had been expected, both the lighting and thermal modules were in clear conflict with each other. Many solutions exist with higher overall lighting performance but at the cost of thermal performance and vice versa. Such conflicts and contradicting objectives are typical for multidisciplinary design problems. The Pareto front of non-dominated solutions offers the design team a good basis to discuss trade-offs between objectives.

However, it should be noted that this study is done based on a single-objective evaluation, and as the plot in figure 9.44 shows, there are undesired gaps in the non-dominated solution front.

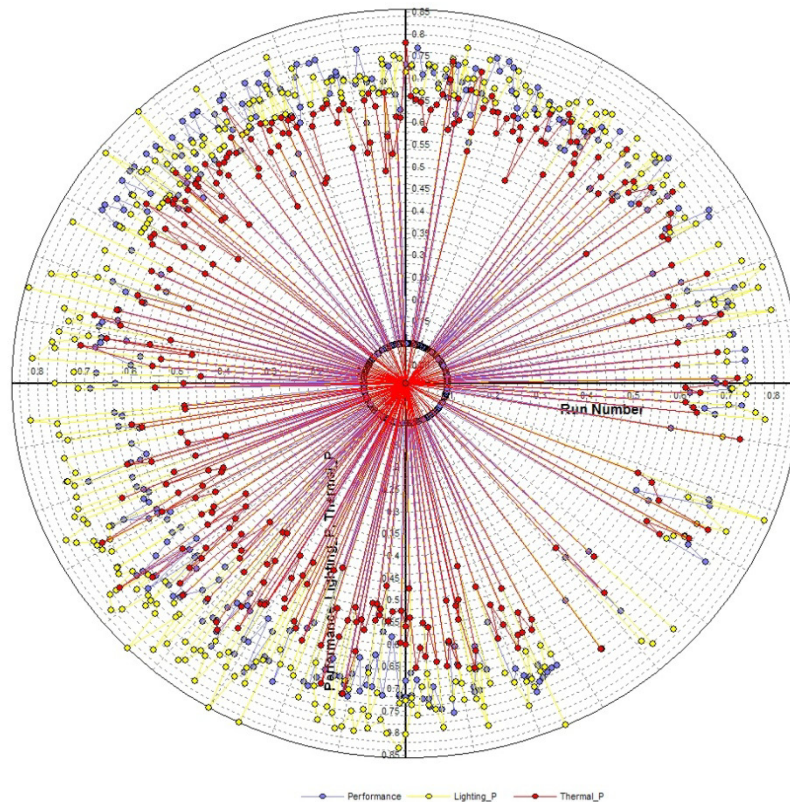
Radial plot was later used to visualize the trade-space (figure 9.45). However, due to the large number of solutions, as well as objectives, the plot was not very useful in identifying the best tradeoff between the different objectives.

The design team was then interested in understanding the possible tradeoff between all seven objectives. For many dimensions the 3D plots are no longer useful and an entirely different mechanism must be used.

The design team decided to use a profile plot. The profile is a simple representation in which the score of each objective is scaled vertically along distinct point on the horizontal axis. The performance of any specific non-dominated solution appears as a zigzag horizontal line.

Figure 9.45:

Radial Plot to visualize the trade-space



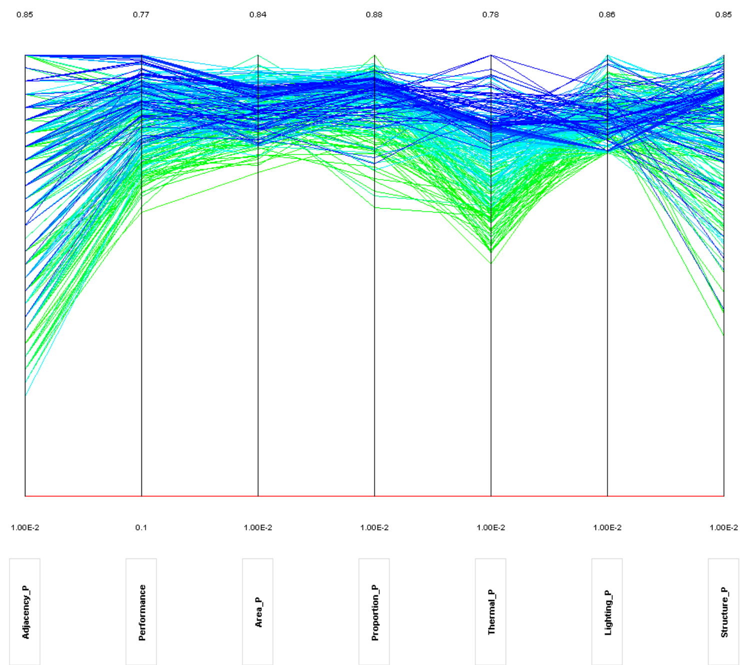
This type of representation was useful initially, but became confusing as more solutions were displayed simultaneously. This mode of presentation is therefore effective only when comparing a few solutions (figure 9.46).

Nevertheless, results from the Pareto filtering proved to be rather interesting, as understanding the trade-offs between conflicting objectives added another dimension towards our ability to interpret results.

After assessing the results of the runs a solution was chosen. Although this solution performed highly in all seven objectives, it was not the one with the highest total performance. This is due to qualitative aspects that were not included initially in the MDDS formulation but the design team believed were important (figures 9.47, 9.48, 9.49, 9.50, and 9.51)

Figure 9.46:

A profile plot of the solutions and the Pareto front. . The performance of any specific non-dominated solution appears as a zigzag horizontal line.



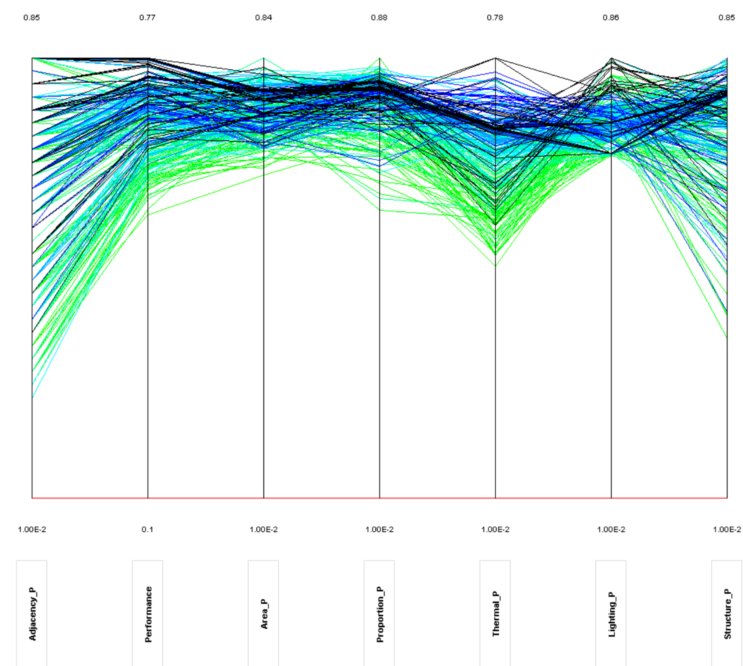
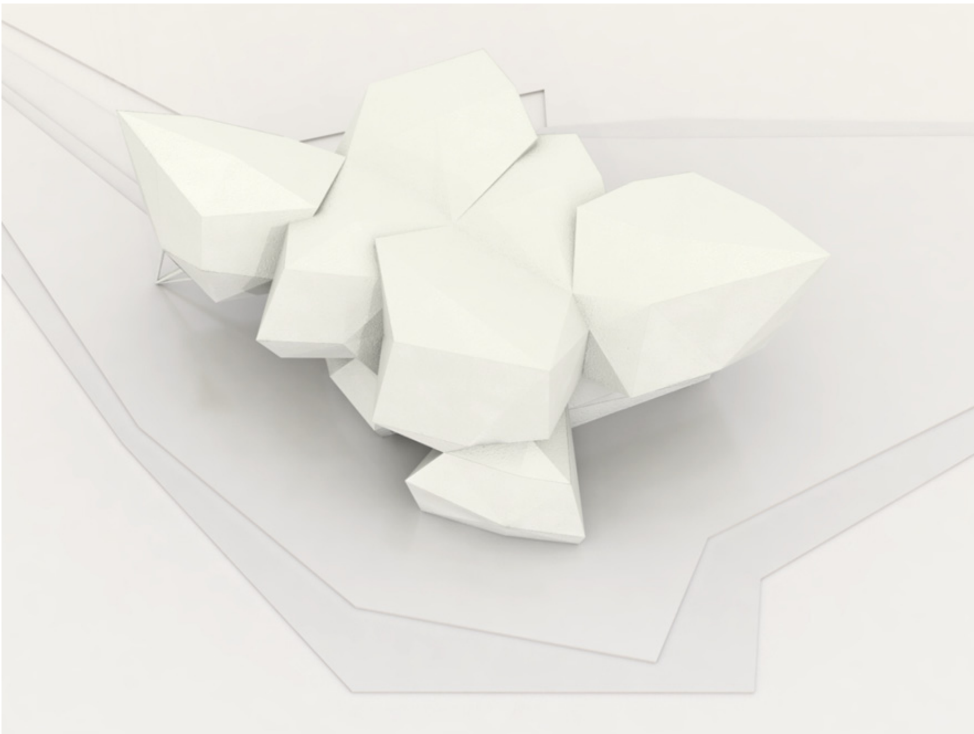


Figure 9.47:

Exterior renderings of chosen solution



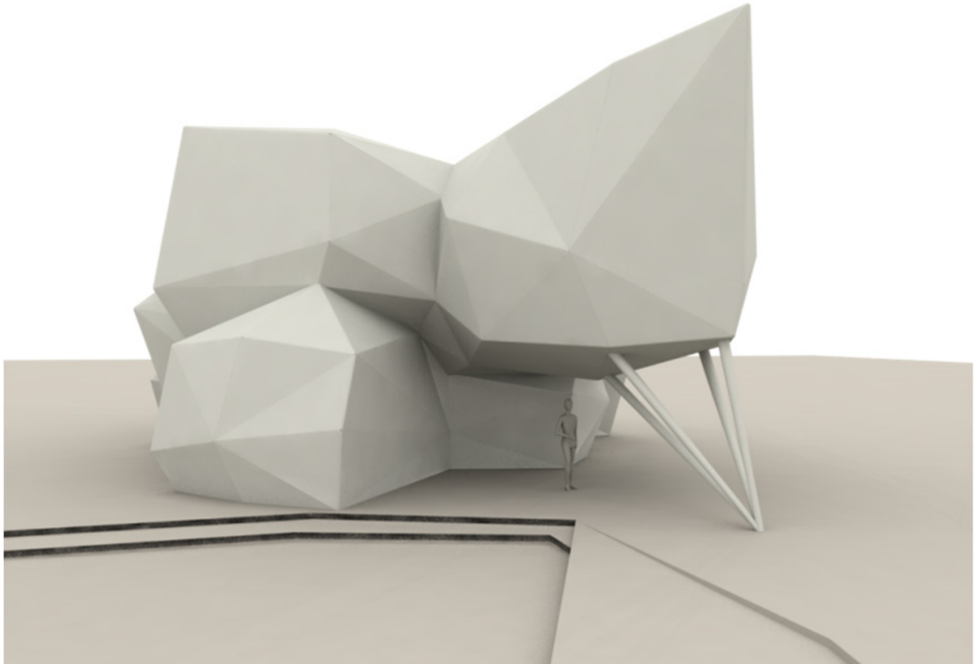
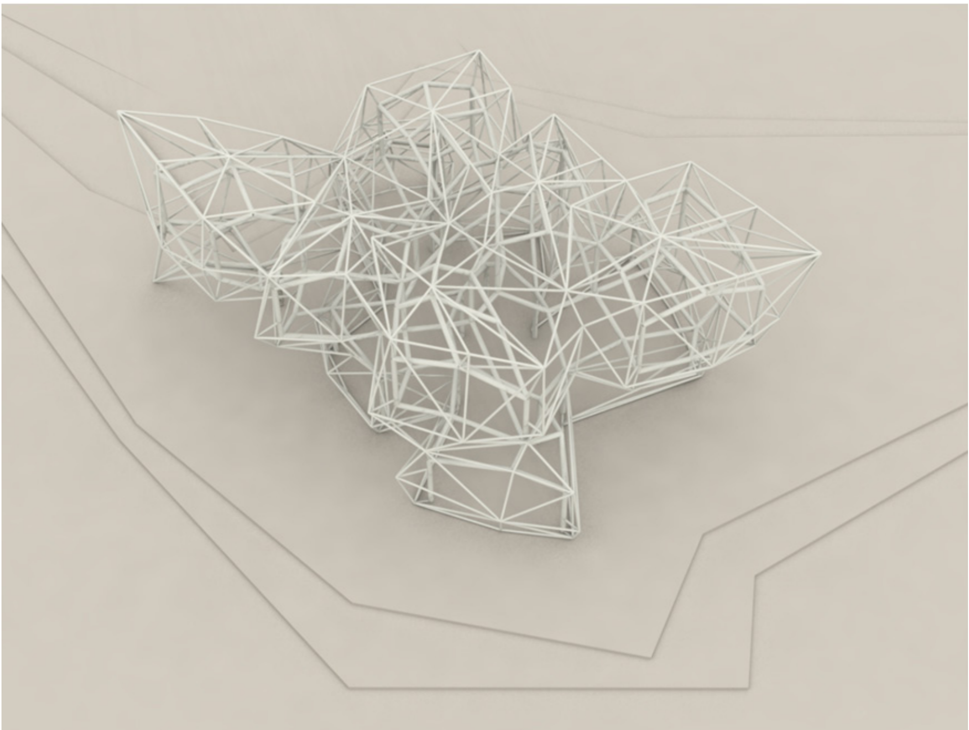


Figure 9.48:

Renderings of structure of chosen solution



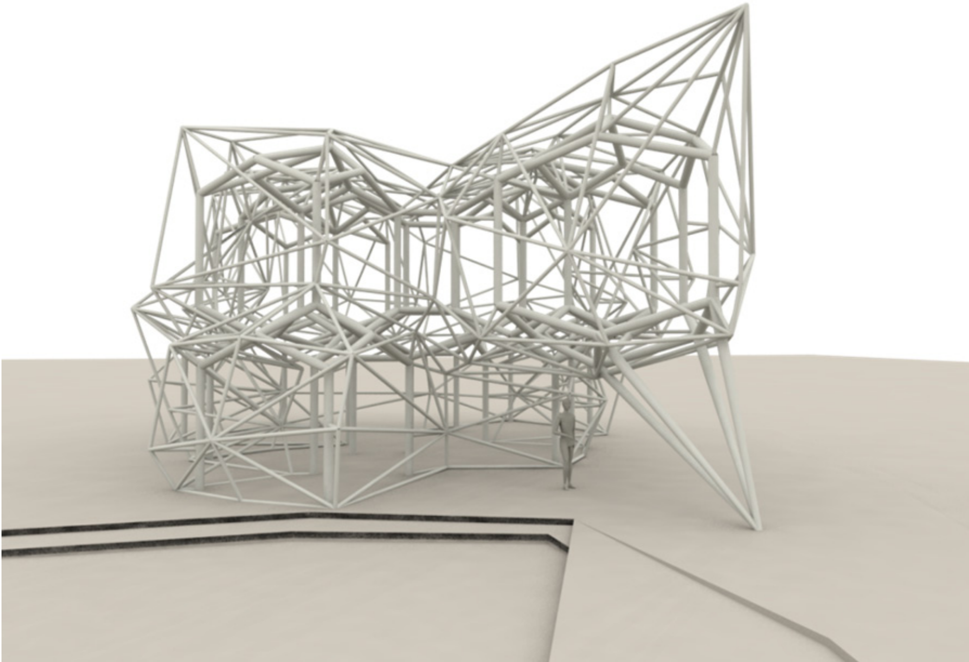
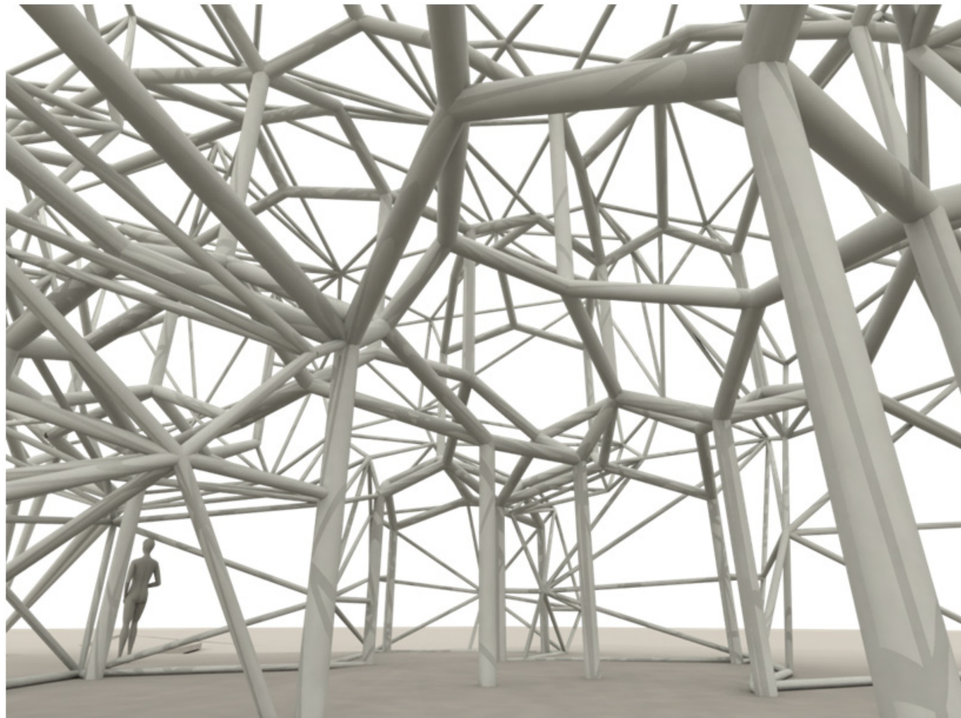


Figure 9.49:

Renderings of the interior of the structure of the chosen solution



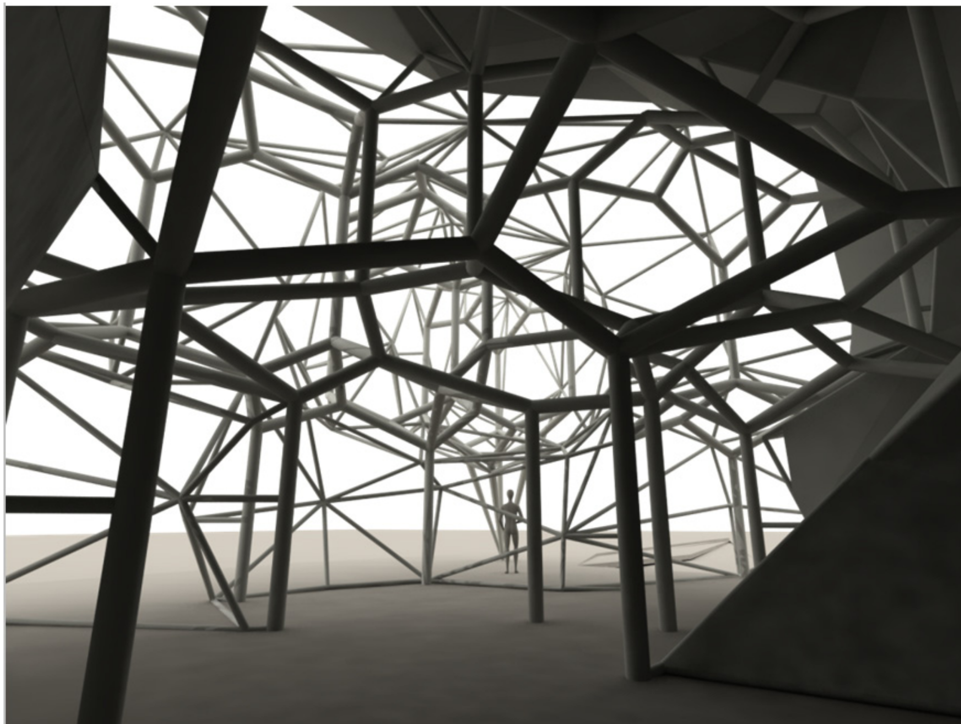
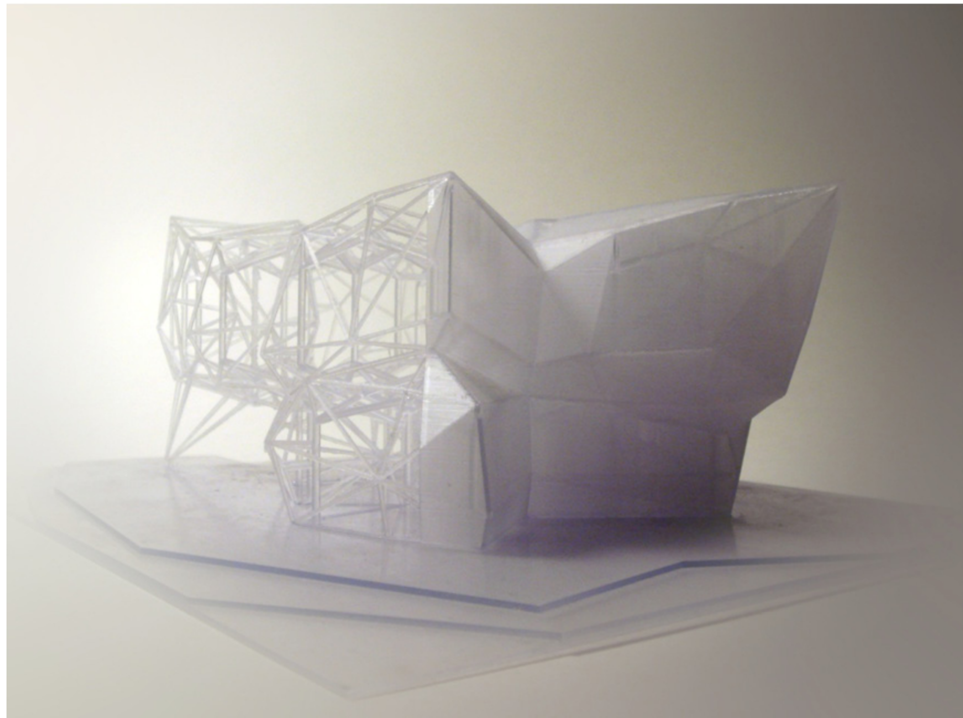


Figure 9.50:

3D Physical model of the chosen solution



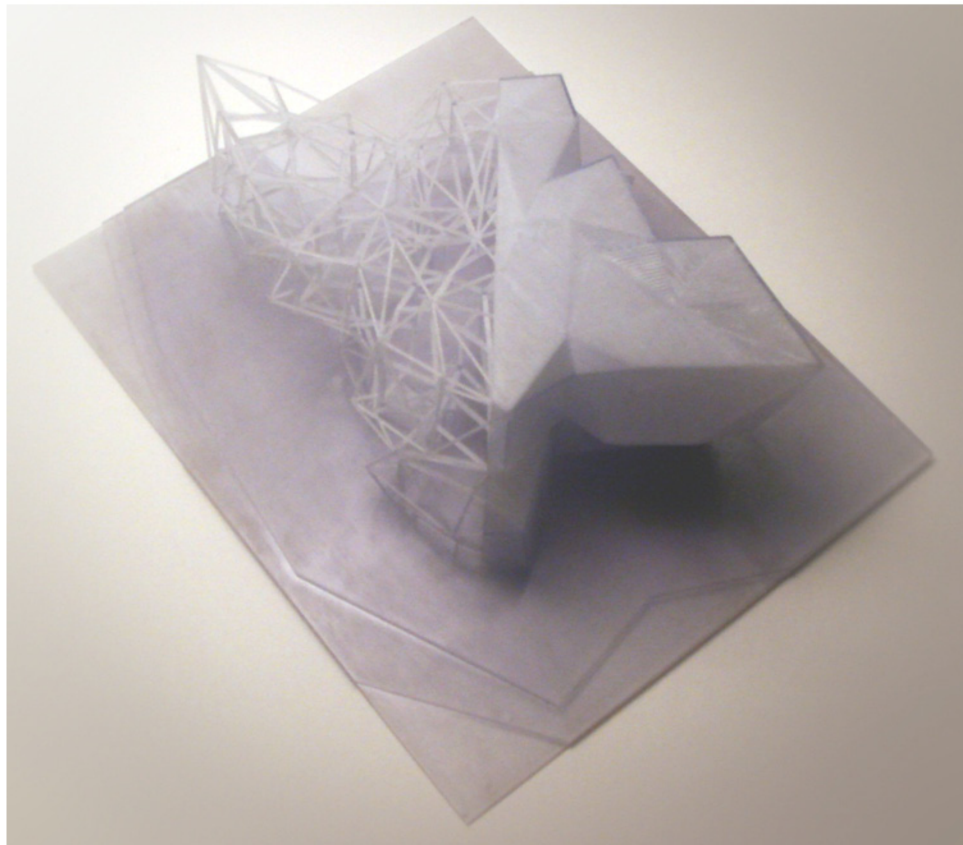
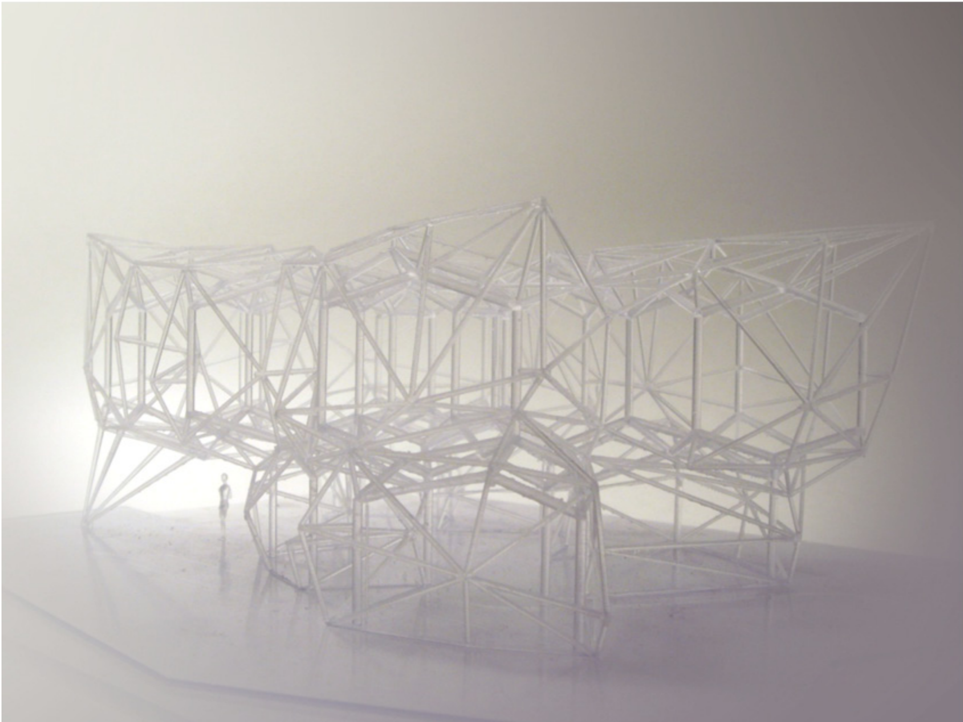
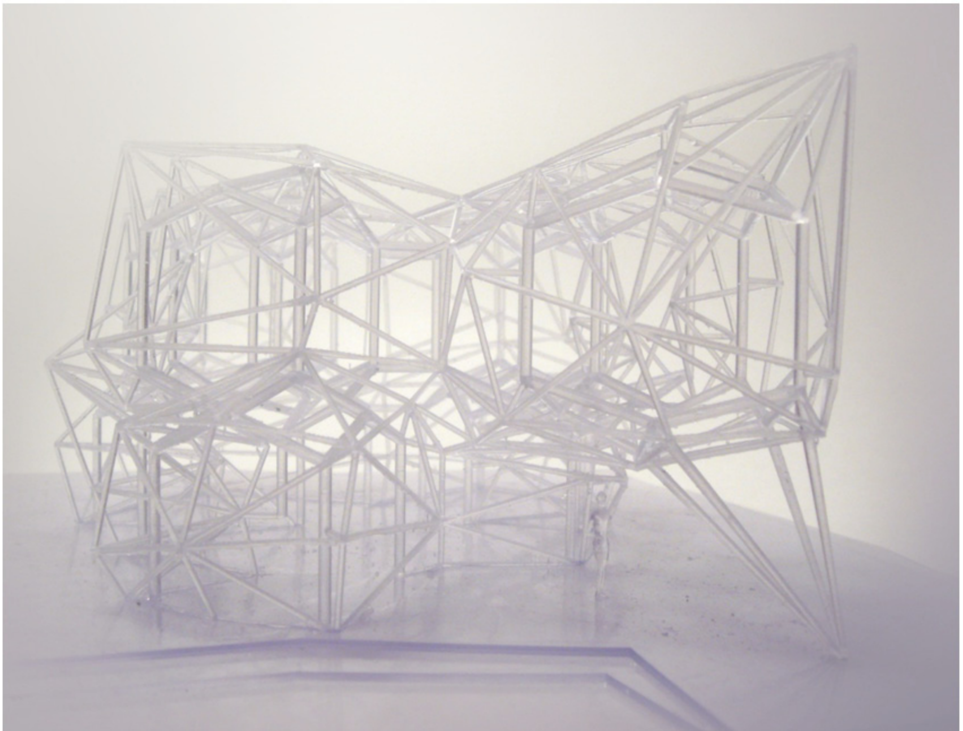


Figure 9.51:

3D Physical model of the structure of the chosen solution





10. Conclusion

10.1 Thesis Summary

In this thesis I discussed different concepts that represent the stages needed to construct what I called the Multi-Disciplinary Design System (MDDS). There are five phases to generate an MDDS. These phases involve decomposition, formulation, modeling, integration, and exploration. These phases are not carried out in a sequential manner, but rather in a continuous back and forth movement between the different steps as the design progresses and evolves.

10.1 .1 Decomposition

As mentioned earlier decomposition, is the first process that takes place at the front end of the MDDS construction development. As discussed earlier, design can be seen as both an object and a process and therefore two modes of decomposition were presented in the thesis, namely object and process decompositions. In object decomposition the artifact or system's design concept is broken down into different components and aspects that make up its physical object. In process decomposition the design concept is broken down into the developmental levels and design activities that can be used to reconstruct the design process.

Process decomposition is required in the MDDS formulation stage in which development decomposition informs the formulation stage about the proposed hierarchy and multilevel structure of the MDDS while activity decomposition is essential in identifying the design activity modules within every level of the MDDS formulation. Furthermore, object decomposition is needed in the MDDS modeling stage were component decomposition is essential for the synthesis mathematical models while aspect decomposition is critical for the analysis mathematical models.

10.1 .2 Formulation

Design process planning follows in the formulation stage which provides an improved understanding of the process properties.

Formulation can be seen as the process of designing the architecture of the MDDS. The MDDS architecture is broken down into hierarchical levels, each of which comprises a group of cycles that include modules at different degrees of abstraction.

The MDDS is broken into hierarchical levels in order to manage the design complexities, where each lower level becomes more detailed and refined as the design progresses. Each module within the MDDS represents a design activity. Similar activity modules can be interconnected to create assemblies. Each cycle within a level includes modules that represent all of the four design activities mentioned earlier, namely synthesis, analysis, evaluation, and optimization. The MDDS as a whole can be seen as a set of interrelated modules that collectively can produce design solutions.

MDDS includes both hierarchical and non-hierarchical structures. The MDDS levels represent a hierarchical structure while relations between the different modules and cycles within an MDDS level represent a non-hierarchical structure.

In the formulation chapter, several tools and notations have been suggested for the task of structuring and formulating the information produced from the decomposition stage into a coherent MDDS architecture. The DSM, for example, could be used to refine the interaction between modules and minimize iterations as well as determining crucial activities that influence process lead-time and cost. Formulation notations that include network notations, such as Data Flow Diagrams, or IDEF0, or even formulation modeling languages such as UML and SysML, can be of great use in designing the MDDS architecture and defining its hierarchical levels, cycles, assemblies and module interactions.

As suggested previously, formulation enables the visualization of data and control flow. Different design processes and architectures can be compared and evaluated. It should be noted, however, that in order to arrive at a reasonable system architecture there must be an iterative cycle or loop between decomposition and formulation.

10.1 .3 Modeling

MDDS offers a framework for modeling design activities that include synthesis, analysis, evaluation and optimization. These design activities are built into modules which can contain mathematical models as well as data or software applications that interact together in order to automate the process of design search.

Each module has a boundary that cuts across its links to the environment defining the module's input and output. Each module acts like a black box transforming data from one form to another. The behavior of each module contributes not only to the design aspect and discipline it is modeled after but to the MDDS as a whole.

Synthesis

The design concept is decomposed into a set of synthesis modules by extracting design intentions and formulating a collection of design parameters, rules or algorithms. These modules define the system components and configurations to be modeled and are based on the component decomposition completed in the decomposition stage.

Synthesis modules provide a representation of the artifact design language which in turn defines the general design space. The design vector is the input to this type of module. As discussed previously in the modeling chapter, the number and type of variables included in the design vector depends on the algorithms and structure of the synthesis module. Synthesis modules output data to analysis modules. This data consists of certain artifact's attributes, such as dimensions, areas, volumes and mass properties. The need for integrating synthesis and analysis modules affects to a great extent modeling requirements for both design activities.

Synthesis models should provide for a generative mechanism. This could be done through the different techniques discussed in the modeling chapter such as parametric and algorithmic models. Parametric models provide for a description of the artifact through parameters and relationships that allow for variation. Algorithmic models give a description of the artifact through a set of rules and algorithms. Generative formal grammars are good examples of algorithmic models. These include grammars like shape grammars, graph grammars, Lindenmayer Systems, and cellular automata.

It was also noted in thesis that the representation of generative synthesis models should encode design knowledge. The relationship between form and performance should be embedded within the representation formalism. This provides restrictions on permitted designs and ensures that the rules discard designs that do not comply with constraints. However, since synthesis models do not include performance feedback loops, it is difficult for such models to direct the generation and navigate the design space of multi-performance design problems.

Analysis

Analysis models and simulations are used to predict the behavior and performance of a specific synthesized design. A design problem usually combines different disciplines, with each discipline developing one or more analysis models.

The outcomes produced by a synthesis module are the inputs to the analysis module. These inputs may range from simple parameters and data, such as areas or volumes, to full CAD models for use by numerical analysis models like FEM and CFD. The outputs of the analysis module are performance measures that will eventually be used within the evaluation modules in assessing the effectiveness of a system configuration.

Analysis models range in their amount of required information input and their degree of accuracy output. Three types of analysis models were discussed in the modeling chapter: analytical, numerical and surrogate models. Analytical models are mainly low-order (low-fidelity) models. Numerical models, like FEA and CFD, are considered high-order (high-fidelity) models which if combined with search and optimization can result in long durations. Surrogate models, such as kriging and response surface models, are low-processing approximation techniques that can be utilized to replace expensive and detailed numerical models. However, these types of models have limited design application.

In choosing an analysis model the design team must select the best compromise between the demand for simplification and the necessity to clearly identify, describe and rate the targeted physical mechanism. A trade-off will have to be made between fidelity and analysis time.

Evaluation

Evaluation models are in essence decision-making tools. The need for the evaluation of results arises when multi-disciplinary objectives exist. The inputs and outputs to an evaluation module depend on the structure of the module, the strategy used in the evaluation and whether it is done before or after optimization.

When the preference is expressed beforehand, the designer decides how to aggregate different conflicting objectives into a single objective function before the search is performed. A commonly adopted approach is scalarization which consists of combining several objectives into one scalar cost function. Different

scalarization methods, such as the weighted-sum approach and the utility function method, were presented in the modeling chapter.

When a search is performed before decision-making, the search is performed with multiple objectives at the same time. The solution space becomes partially ordered with a set of optimal trade-offs between the conflicting objectives called the Pareto optimal set. Several techniques and algorithms for multi-objective optimization, such MOGA, were also presented in the modeling chapter.

Optimization

Optimization models are design space search mechanisms. Searching the design space entails finding the best solution(s) within a domain of feasible solutions. An optimization model seeks to minimize or maximize an objective function by varying the values of the variables in the design vector.

The input to the optimization module is an objective function(s). The outputs of the optimization module are new values for the design vector variables. The choice of an appropriate search algorithm depends on several factors including the design synthesis model, the nature of the analysis models, the number and type of the design variables in the design vector, the existence of constraints, and the linearity of either the objective function or constraints.

As discussed earlier optimization algorithms could be divided into discrete optimization algorithms or heuristic algorithms. Some discrete optimization algorithms that handle constraints include the simplex method, sequential quadratic programming, and the exterior and interior penalty methods among others. Discrete optimization algorithms that handle unconstrained problems are generally gradient-based algorithms. These include Newton's method, steepest descent and conjugate gradient among others. Heuristic algorithms on the other hand include optimization algorithms, such as evolutionary algorithms, simulated annealing and tabu search.

As mentioned previously, no existing optimization algorithm is guaranteed to find the global optimum of a nonlinear, non-convex problem. Gradient-based methods find optima with high reliability but might not escape a local optimum. Heuristic algorithms might find a good solution, but its optimality cannot be guaranteed since they often tend to find a different design each time they are run in addition to the fact that they do not converge to a solution in the same effective manner as gradient-based methods do.

The Design Cycle

In the modeling stage both the design vector variables and the objective function are better defined but can still be modified further according to investigations made in the design exploration stage, either before search or after search.

Domain knowledge of each discipline involved in the design informs the synthesis modules to create meaningful designs and representations. The outcome of the synthesis modules is analyzed by the different discipline analysis modules to predict the properties of a particular solution. The evaluation modules then handle the multi-objective nature of the design. The optimization modules search the design space and automate the synthesis, analysis and evaluation in search of new solutions. The process continues until the optimization has converged and a family of acceptable solutions is found.

10.1 .4 Integration

Integration takes place at the tail end of the MDDS development. Design activity modules that were modeled and created by design specialists are integrated to create an MDDS cycle. Design cycles can also be integrated within a level and so on. Through a bottom-up approach, all the design activity modules that were developed are connected into a data flow network that includes clusters and subsystems.

Integration tools are used to satisfy the requirements of the MDDS process through efficiently automating the exchange of module information. The integration between the different modules can be carried out using one of the integration technologies discussed earlier in the integration chapter such as middleware, web services or a combination of both. As discussed earlier, MDDS also supports the integration of commercial analysis and simulation programs through the automation of program execution. Integration mainly aims at facilitating the integration of design activity modules and simulation programs regardless of discipline or programming language.

The end result of a typical MDDS process is an integrated system model. The MDDS can then be tested to verify that it actually works as formerly planned. Testing the system involves running the simulations and reviewing the model validity.

10.1 .5 Exploration

After building and integrating the MDDS, it would be useful to carry out a few experiments that could help explore the design space. MDDS can be continuously adjusted through several process iterations in order to investigate the influence of the modification of different variables in the design vector.

Changes in the design variables of one part of the system are rapidly spread throughout the system. Design space exploration can delve into “what-if” scenarios and assess trade-off situations. This makes it an essential tool for analyzing the effects of design variables and the shape of design spaces to provide a better understanding of the decisions that are made in design selection and the corresponding consequences. This aids designers in the process of determining the best trade-off among performance and cost, in addition to enhancing multidisciplinary negotiations, leading to better design quality.

Exploration experiments and techniques are not intended as a validation of the system as a whole as much as they are a validation of some of the design decisions made within the MDDS, such as what variables to include in the design vector or the structure of the objective function. As mentioned previously in the exploration chapter, exploration can be carried out before search or after it.

Exploration techniques used before search include methods, such as DOE, that can be used to provide an overview of the design space or a local region of the design space. These techniques can be used to screen factors, thus helping minimize the problem size before the optimization process takes place. In addition, a new and feasible or enhanced initial point for optimization can sometimes be chosen using the initial points analyzed from the DOE study.

Exploration techniques used after search and optimization are mainly sensitivity analysis processes. Sensitivity analysis is primarily concerned with how the specific response of a chosen solution changes due to the modification of design problem formulation. Sensitivity analysis tries to identify what source of uncertainty affects the final solutions more. The importance of sensitivity analysis comes from the fact that all the mathematical models used in the MDDS are approximations to the actual artifact and system.

10.2 Thesis Contributions

Some of the concepts put forth here are not new, but the

contribution of this thesis stems from synthesizing these concepts into a coherent whole. In the following I will discuss some of these concepts in further detail.

10.2.1 A Computational Design System Model

The main idea proposed in this thesis is a framework for developing computational multi-disciplinary design systems (MDDS) that would enhance the design of complex systems and artifacts through an efficient process. This proposed MDDS framework is a generic framework that proposes a group of systematic methodologies that eventually lead to a fully realized and integrated design system.

As stated previously, the MDDS embraces diverse areas of research that include design science, systems theory, artificial intelligence, design synthesis and generative algorithms, mathematical modeling and design oriented disciplinary analyses, optimization theory, data management and model integration, and experimental design among many others.

The hope is that this computational design system can assist the design team in going beyond their limitations in searching huge design spaces. By implementing the framework, vast design spaces can be searched while solutions are intelligently modified, their performance evaluated, and their results aggregated into a compatible set of design decisions.

Within this framework, complexities of the design can be handled and the uncertainty of its evolution can be managed. Furthermore, MDDS techniques provide a better understanding of not only the designed artifact properties, but also of the priorities of the design system expectations and objectives.

As demonstrated throughout this thesis, the way in which the resulting design system is generated and the capabilities it brings are totally different from the tools used in traditional design approaches. The MDDS is therefore more of a design process than a specific design tool or group of design tools.

In addition, the framework presented supports the design of complex systems within a variety of domains. The hope is that by incorporating the MDDS designers can gain a market edge through enhanced design quality and performance and improved collaboration among multi-disciplinary design teams that would lead to reduced design time and cost.

10.2.2 A Multidisciplinary Design System Model

In the traditional design approach, many actors participate in different phases, each with diverse competencies and seeking solutions to a particular aspect of the design problem. In this sequential approach, due to the well-defined boundaries between disciplines, different competencies work on the design at different times, each one modifying the product of the previous one to achieve its objective.

Thus, the final solution is not always the optimal solution or the one that requires the least time to figure out. In fact, the overlapping of many decision-makers who act separately, and often times have conflicting goals, generates recurrent changes and unnecessary feedback loops in the design process. If all the requirements and partial objectives had been taken into account at an earlier stage, a more suitable and economic solution would have been found.

Using MDDS, the design solution is not envisioned a priori, and a very wide exploration of potential solutions is encouraged. Each solution is rated on the basis of multi-objective criteria operating simultaneously.

It is easy to understand the advantages of performance-driven, concurrent design with respect to the traditional sequential approach that, due to the complexity of the design problem, envisions a very limited number of potential solutions, assesses their efficiency and checks for feasibility. The “optimality” of the solution, in that case, relies heavily on the experience of the designers and of the project coordinators, and often lacks the benefits provided by the MDDS integrated approach.

10.2.3 An Evolutionary Design System Model

Design can be seen as an evolutionary process. One of the main attributes of the MDDS framework is that it takes into account this evolutionary nature of design. As discussed earlier, design descriptions change as projects progress. A design cannot be described at the detailed level required for manufacturing at the earliest stages of design. The level of description of a specific design is directly proportional to the amount of information available at a specific project stage. With project design progress and evolution, the complexity of both the design description and the corresponding design models increase as design progresses.

Therefore, the resulting MDDS model is described by an evolutionary

model moving from simple and generic ideas into more complex and detailed ones throughout the process. This notion of an evolving system yields an MDDS that is continuously dependent on, and responsive to, the uncertainties of the design progress. MDDS captures the design evolution through an evolving system architecture. New levels, new cycles as well as new modules are added as the design progresses. MDDS is thus characterized by comprising a multi-level, multi-module, multi-variable and multi-resolution architecture.

Multi-level

Since both the physical artifact and the design process can be viewed in terms of hierarchical decompositions where they are decomposed into multi-levels, the MDDS architecture should also be considered multi-level. The MDDS process should be viewed as an incrementally changing process that grows from the top to bottom as a combination of multiple quasi-interdependent levels. Each level in the MDDS can be decomposed into design-cycles that can be further decomposed into different linked modules.

Multi-module

As discussed earlier, many design problems require using a group of complementary models, rather than one single model, which collectively aim at modeling and describing the whole design problem. This modeling process requires specialized knowledge in many disciplines. MDDS facilitates this by its multi-module platform for utilizing several design activity modules from different disciplines to simulate design problems.

Multi-variable

As the design evolves the set of variables in the design vector also evolves and changes between the different levels of the MDDS. Design variables at a certain level become constants at a lower level. At the same time new variables are added to the design vector at lower levels. This multi-variable property changes the degrees of freedom of the design system from one level to the next.

Multi-resolution

Furthermore, for the evolving MDDS, modules with different resolutions and granularity levels are needed. By altering modules or exchanging existing disciplinary synthesis and analysis modules for more suitable fidelity levels, existing MDDS level modules can be

evolved to lower successive levels. Therefore, MDDS involves a multitude of model resolutions. In conceptual design, low-fidelity models are used in the MDDS due to the lack of complete and sufficient design information. In later phases however, more detail is required to perform elaborate synthesis and analysis. Hence, these are conducted using higher-fidelity models.

Decoupling

Although MDDS starts with integrated and coupled design cycles, these design cycles tend to decouple as the MDDS levels are created and evolved. Decoupling takes place both horizontally and vertically when the interactions between modules or levels disappear. This happens when the various interconnected modules are decomposed into different cycles which do not require as their input the output of another cycle. The system structure is thus simplified and can benefit from parallelism.

10.2.4 An Adaptable Design System Model

The MDDS can adapt to changes in the design evolution process due to the modular nature of the MDDS; many different options can be generated using its modular mix-and-match flexibility. As mentioned earlier, a design module can substitute for another, a new module can be added to the system, a module can be deleted from the system, and a module can be reused in another MDDS.

An existing MDDS level can be evolved in this context to a lower level through changing or replacing existing disciplinary analysis modules within design cycles to more well-suited modules with the adequate fidelity levels. Previously developed modules in another MDDS can also be adapted to the current MDDS.

Furthermore, the modular nature of MDDS facilitates conducting trade studies and affords the design team with greater flexibility in addressing dissimilar and large trade-spaces. Traditionally, conducting a multi-disciplinary trade study is characterized as a time consuming process which is largely dominated by the transforming and translating of data between design disciplines. The MDDS approach would allow the quick interchange of individual modules, leading to easily testing the effect of these modules on the design solutions in addition to customizing different scenarios to the specific problem for effective exploration.

10.2.5 A Generative Performance-Driven Design System Model

One of the important contributions of the thesis lies in incorporating generative synthesis models. As discussed earlier, in most of the work done thus far in other design technologies, such as MDO, the topology of the artifact is generally fixed by the design team and the optimization merely varies its dimensionality. As demonstrated in the experiments, MDDS encourages the use of generative synthesis models that generate more varied design spaces. This enables multidisciplinary design teams to formally explore the performance of many more design alternatives, which should lead consequently to better designs and enhanced performance.

Furthermore, the MDDS approach introduces a scenario where the idea that performance drives design is clearly identified. In today's increasingly competitive market, design solutions that merely meet minimum project requirements are no longer guaranteed to prevail. Solutions must be cost-effective and generated through efficient multi-disciplinary processes. An effective evaluation of these solutions therefore involves the integration of multiple disciplines. MDDS allows for identifying counter-intuitive solutions and functions of multiple design disciplines.

Although MDDS helps generate high performing solutions, these performance measures are mostly driven by quantitative aspects of the design. I believe, however, that qualitative aspects of the design should also be taken into account. A quantitatively optimum solution might not necessarily be the best solution. Better exploration of the design space might reveal solutions with better qualitative merits. That is why the MDDS framework proposes the generation of families of quantitatively good solutions that can later be assessed by the design team for their qualitative aspects.

10.2.6 A Design System Model with Emergent Behaviors

The various modules involved in the MDDS try to optimize the design of the artifact, each within its respective discipline. This clearly creates conflicts between the different design modules. From this conflict, unexpected solutions can emerge.

As mentioned earlier, the MDDS functions as a dynamic and complex whole, interacting as a holistic structured functional unit. The system emergent properties are not detectable through the properties and behaviors of its modules, and can only be enucleated through a holistic approach. The solution found by this system is expected to be superior to the design found by solving and optimizing each

discipline sequentially, since it can exploit the interactions between the different disciplines.

This emergent capability in MDDS is an attractive quality that can address Descartes' Dictum proposed in the introduction of this thesis: "how can a designer build a device which outperforms the designer's specifications?" (Cariani, 1991). By identifying unexpected solutions, the MDDS can help designers reach beyond their manual design limitations, and therefore, arguably, can be described as exhibiting intelligent behaviors.

10.2.7 A Model that Reduces Design Iteration Time

As stated in this thesis, MDDS can significantly minimize design iteration time by implementing several methodologies and technologies that include integrated design approaches as well as automated design activities, leading to enhanced efficiency in design iteration time.

This reduction in time can enable design teams to formally explore the performance of a variety of design alternatives. This is considerably more than is currently possible within the same duration using traditional design approaches. With MDDS more time can be spent in the interpretation of results and in choosing between design alternatives as well as reshaping the design space in search of more promising regions.

10.2.8 A Model that Redefines the Design team and Studio

In addition to the design process, both the design team and workspace are affected by the MDDS approach. The MDDS proposes that the design team should consist of design specialists and system architects that can jointly grasp a large body of knowledge and experience. The role of design specialists is to guarantee that their share of the requirements and constraints in the design process is solved. System architects' role, on the other hand, is to assemble multiple parts of the design process into a full system.

Responsibilities of both the system architect and design specialist can be defined by means of the MDDS hierarchical structure. Modules and sub-cycles denote elements that that lie within the domain of design specialist who can adjust them to a specific application given a group of specifications. The systems architect should be able to manage the complexity of formulating the system architecture. The number of levels, as well as the number and type of activity modules to be included in addition to the technical tradeoffs

that influence the system capabilities, must be resolved by the system architect.

Furthermore, the thesis proposes the need for an MDDS environment that can generate MDDS models by allowing all design participants to embed their specific software tools or models into modules collaboratively and then efficiently integrate these modules into different cycles and levels to create a full MDDS.

Although some interesting commercial tools to manage some aspects of this integration currently exist in the market, these tools still remain limited in the scope of their application.

Given that the MDDS is essentially designed as an assembly of linked programs and components, the workspace within the MDDS can be considered a virtual design studio that implements the component-assembly approach. The MDDS design environment should provide for an infrastructure of data integration tools and methods that supports the robust simulation process for product design and development throughout the design lifecycle. Through this environment, many benefits can be achieved, such as minimized data translations, effective data or knowledge configuration control and architecture, enhanced distributed collaboration by geographically dispersed product teams, and effective data transfer between different stages, from conceptual to preliminary to detailed design. It should also support the integration of commercial and proprietary analysis and simulation programs through flexible coupling methods and automation of simulation program execution. It should also provide for design space exploration using a suite of design space exploration tools.

The system architect, within such a computational environment, becomes a master assembler of digital blocks analogous to the architect within the physical world as the master builder.

10.2.9 A Design System for Integral and Modular Architectures

The MDDS can be considered as a modular system for the creation of varied architectures. This is because regardless of the intended design of the artifact's system architecture, whether modular or integrated, the MDDS can help enhance the artifact's performance. This is due to the fact that, although influencing each other, object decomposition and process decomposition are handled separately within the MDDS. Therefore, the artifact's proposed design object can have either a modular or an integrated architecture, while the

MDDS as a computational design system can remain modular.

10.2.10 A New Approach to Building Civic Architecture

Although the MDDS framework is intended to be domain-independent, several MDDS prototypes were developed within this thesis to generate exploratory building designs. I hope these examples provide a proof of concept for the validity of the framework presented in this thesis and the potential it has on influencing the computational design approach.

10.3. Limitations and Difficulties

Although the MDDS framework provides a powerful approach to designing multidisciplinary systems, there are still several difficulties that need to be investigated and researched further. I will summarize here a few of these difficulties, which include issues related to synthesis complexity, analysis representation, multi-level optimality, evaluation visualization, algorithmic exploration, and setup time.

10.3.1 Synthesis Complexity

Most multi-disciplinary optimization problems and applications involve tens, and even hundreds or thousands, of design variables and constraints. This denotes a significant difficulty for computational design both in managing the design variables and in the ability to search the multi-dimensional design space adequately.

Furthermore, the algorithmic synthesis models discussed in this thesis, such as formal grammars, represent difficulties for both analysis and optimization. This is because these systems are in flux and can change the number of variables that represent a solution as well as the configuration of the solution. There remains a lot of work to be done in computational systems for design before workable methodologies for these types of problems are realized.

10.3.2 Analysis Representation

One of the fundamental challenges with applying the proposed MDDS is the issue of large-scale data management and data representation.

Extracting and transferring the design information from different design models can represent varying difficulties. It can take the form of simply translating the syntax of one program output into another program input. It can get more complex, such as when a generative

synthesis model produces new topologies and geometries at each new iteration. This represents considerable modeling difficulty since, on one hand, the analysis model has to extract new relevant information from the synthesized solution, and, on the other, the optimization has to handle a varying size design vector(s).

These challenges are very distinctive from those that have been addressed by the industry for a long time. More research has to be carried out since methods for dealing with such issues previously have been shown to be insufficient for solving this type of problem.

10.3.3 Multi-Level Optimality

As stated in the thesis, the MDDS proposes a successive filtering of solutions, in which certain solutions, with a certain degree of abstraction, are moved from one level to the next lower level to be optimized further. This is a sequential optimization technique between deferent levels which is not expected to necessarily lead to an optimum solution.

As discussed in the thesis, the question of optimality in a multidisciplinary environment is debatable especially with the existence of qualitative aspects. Several design aspects can only be assessed by the stakeholders and design team when more knowledge about them becomes available as the design evolves.
One of the difficulties associated with optimization in MDDS is the uncertainty involved within the design process as a design evolves from one level to the next. In many cases, the lower levels will not be known. However, after the full design has evolved and further optimization of the full system is sought, several multi-level optimization techniques may be implemented although with a limited scope due to the complexity of the design vectors involved.

10.3.4 Evaluation Visualization

As was demonstrated in the experiments, the visualization techniques of the objective space and Pareto front were limited to a few objectives. New visualization techniques need to be developed to help the design team and stakeholders understand the trade-off possibilities better.

10.3.5 Algorithmic Exploration

Not many exploration techniques were implemented in the design experiments in this thesis. This is due to type of problem exploration techniques are designed for. Exploration techniques are generally

designed for a parametric representation and not algorithmic representation like the ones used in formal grammars.

New techniques that can gauge and explore a design space generated by an algorithmic representation need to be researched further. These challenges are very distinctive from those that have been addressed by designers and engineers for a long time. More research has to be carried out since methods for dealing with such issues previously have been shown to be insufficient for solving existing problems.

10.3.6 Setup Time

The MDDS efficiency in searching the design space and producing several solutions does not come without a price, particularly when it comes to setup time. A great deal of initial investment is required in setup time and process planning. The hope is that this investment can be compensated throughout the design life cycle.

This investment could be reduced if modules are reused between different design projects. Furthermore, if different modules are made publicly available, such as on the web, time spent on modeling can be shortened.

However, this approach may present its own set of drawbacks. Designers and engineers may be inclined to use certain modules because of their availability rather than their suitability to the design problem. Furthermore, modules may be used without fully understanding their functionality and hence may jeopardize the validity of the design system.

Figures List

1.1 Aircraft Design Optimization Framework Using MDO. Adopted from Martins, J. MDO Lab, University of Toronto.

1.2 MDO Framework for Blended Wing Body Concept (de Weck and Willcox, 2005).

1.3 Proposed Framework for the Multidisciplinary Design System (MDDS).

2.1 Design can be considered both as an object and a process.

2.2 The proposed design system should imitate the design process to produce the artifact.

2.3 Four design domains in the axiomatic design (Suh 1990).

2.4 Zigzagging process between functional and physical domains.

2.5 Functional domain and physical domain hierarchies.

2.6 Short conception design phase with unequal distribution of improved quality and integrated disciplines for optimization.

2.7 Life cycle-cost committed versus incurred by life-cycle phase.

2.8 An example of a building skin component. Knowledge about the design is increased as the design evolves over time.

2.9 At the creative end of the spectrum, design is very fuzzy. As it moves to routine design, it gets precise, crisp, and predetermined (Bahrami and Dagli, 1994).

2.10 The boundary around an office building system determines its relation with the environment.

2.11 The black box approach identifies system performance in terms of inputs and outputs.

2.12 In modular architecture there is a close match between the functional and physical hierarchies.
In Integral architecture functions are distributed among a variety of elements.

2.13 An Example of two Building Skins with one representing a modular architecture and the other representing an Integrated architecture.
Project Credit of Integrated Architecture Skin: Anas Alfaris, Alexandros Tsamis.

2.14 Modular Architecture in physical product design is not always superior to integrated architecture as is illustrated in this nail clipper example (Ulrich, 1995).

2.15 The structure of complex systems can have multiple hierarchical levels.

2.16 A tree with 8 nodes and 7 edges or links, 5 paths from root node to bottom or leaf nodes, 3 levels (Magee et al, 2006).

2.17 Non-standard trees: an impure relatively complex tree with non-standard interconnections (Magee et al, 2006).

2.18 Layered hierarchies with horizontal interconnections (Magee et al, 2006).

2.19 Three layers, a root node, 10 nodes, with horizontal interconnections (Magee et al, 2006).

2.20 Networks with the same topology.

2.21 There are many types of networks which depend on the type of edges connecting vertices (Hayes, 2000).

2.22 Taxonomy of networks (Magee et al, 2006).

3.1 Alexander's representation of the design problem as a network.

3.2 Hierarchical and Network decompositions.

3.3 Design decomposition can consist of object and process decompositions. Object decomposition includes Component and Aspect decompositions, while Process decomposition includes Development

and Activity decompositions

3.4 Power train component decomposition.

3.5 Office building component decomposition.

3.6 Office building aspect decomposition.

3.7 Design development can be decomposed into several stages.

3.8 RIBA's four phase model which includes: assimilation, general study, development, and communication.

3.9 Archer's model includes six types of design activity: programming, data collection, analysis, synthesis, development and communication.

3.10 Archer's reduced model with three broad phases: analytical, creative and executive phases.

3.11 Eggert's design model includes four basic phases: formulating, generating, analyzing, and evaluating.

3.12 March's PDI production, deduction, induction model.

3.13 The Function–Behavior–structure (FBS) Framework Gero (1990).

3.14 Design activity model. Four phases are included: Synthesis, Analysis, Evaluation, and Optimization.

3.15 Asimow's design model includes a vertical and a horizontal structure. (Mesarovic, 1964).

3.16 The Markus/Maver model includes a decision sequence and a design process.

3.17 The city car project demonstrates that a design can have several decomposed views.

3.18 In this school project several decomposition views are produced simultaneously.
Project Credits: Anas Alfaris, Kenneth Namkung, Meredith Elbaum.

4.1 Decomposition breaks a system into components whereas formulation puts them together.

4.2 There are structural patterns pertinent to each

problem (Chermayeff and Alexander, 1963).

4.3 Issues that share many connections are grouped together (Chermayeff and Alexander, 1963).

4.4 An activity-based DSM for the development of a soda bottle (McCord 1993).

4.5 Application of a quantification scheme in a DSM (Pimmler and Eppinger, 1994).

4.6 Four different types of DSM (Browning, 1998).

4.7 Activity information flow and their equivalents (Eppinger, 1991).

4.8 Activity information flows and their corresponding DSM equivalents (Browning, 1998).

4.9 Data Flow Diagram (DFD).

4.10 Function Flow Block Diagrams (FFBDs).

4.11 Characteristics of the model element design review (Andersson et al., 1998).

4.12 Characteristics of the model element design review (Andersson et al., 1998).

4.13 Design development process (Andersson et al., 1998).

4.14 Sample Enhanced FFBD (Long, 2002).

4.15 Sample class object in a class diagram (Pender , 2002).

4.16 A complete class diagram (Pender , 2002).

4.17 A component diagram shows interdependencies of various software components the system comprises (Pender , 2002).

4.18 Deployment diagram (Pender , 2002).

4.19 Sample use-case diagram.

4.20 Activity diagram with 3 swimlanes (Pender,2002).

4.21 Statechart diagram showing the various states that classes pass through in a functioning system (Pender, 2002).

4.22 A sample sequence diagram (Pender,2002).

4.23 SysML diagram taxonomy (OMG,2007b).

5.1 Block diagram representation of a mathematical model.

5.2 Expected input and output of the synthesis model.

5.3 Koch curve is a recursive synthesis algorithm.

5.4 An example of a synthesis algorithm that generates variations of components that create a structure.

5.5 A space truss is generated using a set of parameters.

5.6 A building skin generated from a set of synthesis rules and an algorithm that generates a Voronoi diagram.

5.7 A building skin structure and materiality generated using a set of L-System rules.

5.8 The graph formalism (Alber, 2002).

5.9 Graph Rules (Alber, 2002).

5.10 Graph generation by a production system (Alber, 2002).

5.11 Expansion of a grammatically defined sentence and the corresponding object (Alber, 2002).

5.12 Rule 30 cellular automaton.

5.13 Rule application of a CA.

5.14 Prairie-style house shape grammar (Koning and Eizenberg, 1981).

5.15 In this composition a designer might pick the upper square, the lower one, or the one generated by their intersection.

5.16 A set of Shape Grammar rules can generate many variations of a component.

5.17 Various NURBS can be constructed by the same number of control points with various degrees.

5.18 An enumeration tree for an ingress-egress system Project credits: Anas Alfaris, Nii Armar and Martin McBrien.

5.19 Various synthesis models can generated different solution spaces.

5.20 An unrestrictive model may be able to span several feasible solution spaces.

5.21 Expected input and output of the analysis model.

5.22 Analysis models vary based on their mathematical nature.

5.23 Simple analytical models are used to assess the behavior of a building skin.

5.24 Simple analytical models are used to assess the behavior of a building skin.
Project Credits: Anas Alfaris and Alexandros Tsamis.

5.25 A high-fidelity analysis model for day-lighting is used to assess the lighting quality in different spaces.

5.26 (a) Finite difference and (b) finite element discretizations of a turbine blade profile (Huebner et al., 2001).

5.27 Finite Element Analysis of different components in this vehicle egress and digress system.
Project credits: Anas Alfaris, Nii Armar and Martin McBrien.

5.28 CFD Model for a building site to study the airflow around the building.
Project Credits: Anas Alfaris, Kenneth Namkung and Meredith Elbaum.

5.29 Taylor series uses localized derivative information at point *Xo.*

5.30 Polynomial fitting uses information from different points for each curve.

5.31 A diagram for a single neuron on the left and a neural net on the right (Papalambros and Wilde, 2000).

5.32 A neural net and three other polynomials modeling the same data (Papalambros and Wilde, 2000).

5.33 A kriging model for a two dimensional function. Two top plots are the actual function and the two lower plots are the kringing model (Papalambros and Wilde,

2000).

5.34 Within an optimization both high-fidelity and low-fidelity models can be utilized.

5.35 Expected input and output of the analysis model.

5.36 Illustration of design space and objective space.

5.37 Sequential variation of weighting factors can be used to find trade-off solutions.

5.38 Different utility functions classifications (Cook 1997, Messac 2000).

5.39 Points A,B,C and D are optimal solutions that are not dominated by any other solution in the search space.

5.40 Expected input and output of the optimization model.

5.41 An optimization problem has an objective function and can have several constraints to insure feasibility.

5.42 A simple taxonomy of optimization algorithms discussed in the thesis.

5.43 Linear program in two dimensions with solution at $x^{*}=c^{T}x$.

5.44 In the steepest descent the trajectory to the solution follows a zigzag pattern.

5.45 The difference between a local minimum and a global minimum.

6.1 Interfaces between different modules have to be compatible.

6.2 Matching interfaces between modules with many variables can be a difficult task.

6.3 In ModelCenter, components are displayed as icons while links are displayed as lines between the components.

7.1 Exploration should be carried out before and after Search and Optimization.

7.2 Multiple combinations of factors and levels are used to analyze the design space.

7.3 An example of a Latin hypercube sampling.

8.1 MDDS Framework includes five phases: decomposition, formulation, modeling, integration, and exploration.

8.2 MDDS application in relation to Duvvuru design categories.

8.3 Object decomposition includes both component and aspect decompositions while process decomposition includes both development and activity decompositions.

8.4 MDDS is broken into hierarchical levels.

8.5 MDDS comprises a group of modules.

8.6 Each design cycle resides in a design level within the MDDS.

8.7 A design cycle can include sub-cycles.

8.8 The MDDS evolves and grows over time, either vertically by adding more levels or horizontally by adding more modules and cycles.

8.9 A design cycle that regenerates a design concept should include synthesis, analysis, evaluation, and optimization activities.

8.10 The different design activity modules are integrated in the MDDS.

8.11 MDDS captures design evolution.

8.12 A guided missile is conceived from the perspective of each design specialist individually.

8.13 Knowledge domains of systems engineer and design specialist.

8.14 Responsibilities of the system architect and design specialist intersect.

9.1 The formalism of the design concept shows five spatial components with interrelations between them wrapped by a skin.

9.2 Component Decomposition.

9.3 Aspect Decomposition.

9.4 Development Decomposition.

9.5 Component and aspect decomposition mapping.

9.6 The MDDS cycle on level one.

9.7 Three rule sets define the synthesis grammar.

9.8 The sequence of application of the three rule sets. Starting with the first cell at time t =0 and ending with the last cell at time t =8.

9.9 The analysis phase includes six analysis modules.

9.10 Adjacency score amplifying factors *g*.

9.11 Variation of k_q (and k_L) with orientation.

9.12 The function $J_{ther}(q)$.

9.13 Graphical representation of the geometric constraints.

9.14 MDDS Module Integration.

9.15 The evolution of solutions. Solutions in the final runs tend to be more compact in their shape.

9.16 CAD models of the design concept.

9.17 Component decomposition.

9.18 Aspect decomposition.

9.19 Development decomposition.

9.20 Component and aspect decomposition mapping.

9.21 Design cycle one and its design activities.

9.22 Structure of the N^2 Diagram.

9.23 The MDDS cycle on level two.

9.24 The distribution of materials and control Points on the Skin.

9.25 Graphical representation of the geometric constraints.

9.26 Structure of the utility functions.

9.27 Integration between different software.

9.28 Interface that plays back Evolution History.

9.29 Experiment#1 Evolution Of Design Using SQP.

9.30 Experiment#1, Evolution of Design using GA's.

9.31 Experiment#2, Evolution of Design.

9.32 Experiment#3, Evolution of Design.

9.33 A Pareto front based on the thermal and lighting objectives.

9.34 A graphical rendering of a solution.

9.35 Diagrams explaining the changes in the Design Concept.

9.36 Component decomposition.

9.37 Aspect decomposition.

9.38 Development decomposition.

9.39 Component and aspect decomposition mapping.

9.40 Design cycle one and its design activities.

9.41 The MDDS cycle on level one, showing extra modules and connections added.

9.42 MDDS Module Integration.

9.43 The MDDS evolution of solutions.

9.44 Pareto front of non dominated solutions.

9.45 Radial Plot to visualize the trade-space.

9.46 A profile plot of the solutions and the Pareto front. The performance of any specific non-dominated solution appears as a zigzag horizontal line.

9.47 Exterior renderings of chosen solution.

9.48 Renderings of structure of chosen solution.

9.49 Renderings of the interior of the structure of the chosen solution.

9.50 3D Physical model of the chosen solution.

9.51 3D Physical model of the structure of the chosen solution.

Tables List

4.1 A taxonomy of types of system element interactions (Pimmler and Eppinger, 1994).

4.2 Scale used to represent different interactions (Pimmler and Eppinger, 1994).

5.1 Analogies between formal and graph languages (Alber, 2002).

5.2 Different Scalarization and Pareto Methods (de Weck, 2004).

7.1 In a parametric study one factor is changed at a time while keeping all other factors at a base level.

7.2 Experiments can be represented in a matrix where each row corresponds to one experiment and each column corresponds to one factor.

7.3 In Fractional designs levels are specified for each factor and outputs are evaluated at every combination of values.

7.4 In the balancing property, for any pair of columns, all combinations of factor levels occur an equal number of times.

9.1 Adjacency requirements.

Bibliography

Aarts, E., Korst, J. and van Laarhooven, P. (1997). "Simulated Annealing". In: Aarts, E. and Lenstra, J. K. (Eds.). *Local Search in Combinatorial Optimization*. pp. 91-120. Wiley.

Abraham, A., Jain, L. and Goldberg, R. (Eds.) (2005). *Evolutionary Multi-Objective Optimization: Theoretical Advances and Applications.* Springer Science, New York, New York.

Abrahamson, S., Wallace, D., Senin, N., and Sferro, P. (2000). "Integrated Design in a Service Marketplace", *Computer Aided Design,* 32(2), pp. 97-107.

Adler, M. Davis, A. Weihmayer, R. and Worrest, R. (1989). "Conflict-resolution strategies for non-hierarchical agents". In: Research Notes in Artificial Intelligence, *Distributed Artificial Intelligence*, Morgan Kaufmann, Palo Alto, CA.

Agre, P. E. (2003). "Hierarchy and History in Simon's Architecture of Complexity". *Journal of the Learning Sciences*. Lawrence Erlbaum Associates. 12(3), pp. 413 – 426.

Alber, R. Rudolph, S. and Krsplin, B. (2002). "On Formal Languages in Design Generation and Evolution. Proc". 5th World Congress on Computational Mechanics (WCCM V), University of Vienna, Vienna, Austria.

Alexander, C. (1964). *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, Massachusetts.

Alexander, C. Ishikawa, S. and Silverstein M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. New York, New York.

Allen, T. F. (1998). "A summary of Principles of Hierarchy Theory". ISSS Atlanta Conference.

Allison JT., Kokkolaros M., Zawislak M. and Papalambros PY. (2005). "On the use of analytical target cascading and collaborative optimization for complex system design". In: Proceedings of the 6th world congress on structural and multidisciplinary optimization, Rio de Janeiro, Brazil.

Anderl, R. and Mendgen, R. (1996). "Modelling with constraints: theoretical foundation and application". *Computer-Aided Design*. 28(3). pp. 155-168.

Andersson, J. Pohl, J. and Eppinger, S. (1998). "A Design Process Modeling Approach Incorporating Nonlinear Elements". Proceedings of ASME Design Theory and Methodology Conference. Atlanta.

Antoniou, A. Lu, W. Murray, W. and Wright, M. (2007). *Practical Optimization. Springer.* New York, New York.

Archer, B. (1984). "Systematic Methods for Designers". In: Cross, N. (Ed.). *Developments in design methodology*. John Wiley & Sons, Chichester, UK.

Ashby, W. R. (1952). "Can a Mechanical Chess-Player Outplay Its Designer?" *The British Journal for the Philosophy of Science.* 3(9). pp. 44-57.

Asimow, M. (1962) *Introduction to design*. Prentice-Hall. Englewood Cliffs, N.J.

Atherton, C. (2002). "An Approach to Multidisciplinary Design, Analysis & Optimization for Rapid Conceptual Design". AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization. AIAA, Atlanta Georgia.

Averill, L. (2006). *Simulation Modeling and Analysis*. McGraw-Hill. New York, New York.

Bahrami, A. and Dagli, C. (1994). "Design Science". In: Dagli, C. and Kusiak, A (Eds.). *Intelligent Systems in Design and Manufacturing*. ASME Press, New York.

Balabanov, V. and Venter, G. (2004). "Multi-Fidelity Optimization with High-Fidelity Analysis and Low-Fidelity Gradients". 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. Albany, New York.

Baldwin, C. and Clark, K. (2000). *Design Rules, Vol. 1: The Power of Modularity*. MIT Press. Cambridge, MA.

Ball, P. (2001). *The self-made tapestry: pattern formation in nature*, Oxford University Press.

Batty, M. (2005). *Cities and Complexity: Understanding Cities with Cellular Automata, Agent-Based Models, and Fractals*. Cambridge, MA, MIT Press.

Bentley, P. (1999). *Evolutionary Design by Computers***.** San Francisco, Morgan Kaufmann.

Bentley, P. and Kumar, S. (1999). "Three ways to grow designs: a comparison of embryogenies of an evolutionary design problem". In: Banzhaf, W. Daida, J. Eiben, A. Garzon, M. Honavar, V. Jakiela, M. and Smith, R. (Eds.). Genetic and Evolutionary Computation Conference, Orlando, FL, p. 35-43.

Black, I. (1990). "Embodiment design: Facilitating a Simultaneous Approach to Mechanical CAD". *Computer-Aided Engineering Journal***.** 7(2), pp. 49-53.

Blanchard, B. and Fabrycky, W. (1990). *System Engineering and Analysis.* Prentice Hall, Englewood Cliffs, New Jersey.

Blazek, J. (2001). *Computational Fluid Dynamics: Principles and Applications*. Elsevier. New York, New York.

Bletzinger, K. and Lähr, A. (2006). "Prediction of interdisciplinary consequences for decisions in AEC design processes". ITcon. 11, Special Issue Process Modelling, *Process Management and Collaboration*, pp. 529-545.

Bletzinger, K-U. and Lähr, A. (2006). "Prediction of Interdisciplinary Consequences for Decisions in AEC Design Processes", *ITcon*, Vol. 11, 529-545.

Bobrow, D. (1984). "Qualitative Reasoning about Physical Systems: An Introduction". *Artificial Intelligence*. 24(1-3), pp. 1-5.

Boggs, P. and Tolle, J. (1995). "Sequential Quadratic Programming". *Acta Numerica*. pp. 199–242.

Bowcutt, K., Kuruvila, G. and Follett, W. (2004). "Progress Toward Integrated Vehicle Design of Hypersonic". 24th International Conferences of the Aeronautical Sciences (ICAS 2004), Yokohama, Japan.

Britt, D. (2000). *Durand Precis of the Lectures on Architecture: With Graphic Portion of the Lectures on Architecture*. Getty Publications, Los Angeles, California, U.S.A.

Brown, A. (1998). "Tool support for enterprise scale CBD: determining your organization's future competitiveness". *Component Strategies Online*. 1(3), pp.31-45.

Browning, T. (1998). "Modeling and Analyzing Cost, Schedule, and Performance in Complex System Product Development". Ph.D. Dissertation. Massachusetts Institute of Technology. Cambridge, Massachusetts.

Buffa, E.S., Armour, G.S. and Vollman, T.E. (1964). *Allocating Facilities with CRAFT*, Harvard Business Review 42(2): 136-140.

Cagan J. (2001). "Engineering Shape Grammars". In: Antonsson E.K. and Cagan J. (Eds.). *Formal Engineering Design Synthesis*. Cambridge, Cambridge University Press.

Cariani, P. (1991). "Emergence and Artificial Life". In: Langton, C. G. and Taylor, C. and Farmer, J. D. and Rasmussen(Eds.), *Artificial Life II, Santa Fe Institute Studies in the Sciences of Complexity*. Vol. X., S. Addison-Wesley. Redwood City, CA. pp. 775-797.

Carpenter, W. and Barthelemy, J. (1992). "A Comparison of Polynomial Approximations and Artificial Aeural Nets as Response surfaces". A collection of technical papers: the 33rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference. Dallas, Texas.

Carty A. and Davies C. (2004). "Fusion of Aircraft Synthesis and Computer Aided Design", 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA-2004-4433, Albany, New York, Aug. 30-1.

Carty, A. (2002). "An Approach to Multidisciplinary Design, Analysis & Optimization for Rapid Conceptual Design". AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, AIAA.

Channon, A. D. and R. I. Damper (1998). "The Evolutionary Emergence of Socially Intelligent Agents". In: Edmonds, B. and Dautenhahn, K. (Eds.), Socially Situated Intelligence: a workshop held at SAB'98, University of Zurich Technical Report. Zurich. pp. 41–49.

Channon, A., Damper, R. (1999). "The Evolutionary Emergence of Socially Intelligent Agents". *In: Proceedings of SAB99 Conference of the Society for Adaptive Behaviour*. MIT Press, Zurich.

Chermayeff, S. and Alexander, C. (1963). *Community and Privacy: Toward a New Architecture of Humanism*. Doubleday Anchor Books. New York, New York.

Chomsky, N. (2002). *Syntactic Structures*. Walter de Gruyter. New York, New York.

Choudhary, R., Malkawi, A., and Papalambros, P. Y. (2005). "Analytic Target Cascading in Simulation-based Building Design". *Automation in Construction*, Vol. 14, No. 4, pp. 551–568.

Coello, C. (2001). "A short tutorial on evolutionary multi-objective optimization". In: Proceedings of 1st International Conference on Evolutionary Multi-Criterion Optimization. pp. 21–40.

Cohon, J. (1978). *Multi-objective Programming and Planning*. Academic Press, New York, New York.

Cook, H. (1997). *Product Management: Value, Quality, Cost, Price, Profits, and Organization*. Chapman & Hall.

Corne, D. and Bentley, P. (2002). *Creative Evolutionary Systems*. San Francisco, Morgan Kaufmann.

Coyne, R. D. Rosenman, M. A. Radford, A. D. Balachandran, M. and Gero, J. S. (1990). *Knowledge-Based Design Systems*. Reading: Addison-Wesley.

Crawley, E. (2003). Lecture Notes for ESD.34 System Architecture. MIT, Cambridge, MA.

Crawley, E. de Weck, O. Eppinger, S. Magee, C. Moses, J. Seeing, W. Schindall, J. Wallace, D. and Whitney, D. (2004). "The Influence of Architecture in Engineering Systems". In: *Engineering Systems Monographof the Engineering Systems Symposium*. MIT, Cambridge, MA.

Cross, N. (1989) *Engineering Design Methods*. Chichester, John Wiley & Sons Ltd.

Cunningham, T. (1998). Chains of Function Delivery: A Role For Product Architecture In Concept Design. Ph.D. Dissertation. MIT.

Czitrom, V. (1999). "One-Factor-at-a-Time versus Designed Experiments". *The American Statistician*. 53 (2), pp. 126–131.

Daffa', *A. (1977). The Muslim Contribution to Mathematics. Croom Helm, London.*

Dantzig, G. (1998). *Linear Programming and Extensions*. Princeton University Press. Princeton, New Jersey.

Dasgupta, S. (1989). "The Structure of Design Processes". In: Yovits M.C. (ed.). *Advances in Computers*.Academic Press. New York. 28, pp. 1-67.

Davis, K. and Bigelow, H. (2002). *Motivated metamodels:Synthesis of cause-effect reasoning and statistical modeling*, The RAND Corporation, Santa Monica,CA.

De Neufville, R. (1990). *Applied System Analysis*. McGraw-Hill College.

De Neufville, R. de Weck, O. Frey, D. Hastings, D. Larson, R, Simchi-Levi, D., Oye, K., Weigel and A. Welsch, R. (2004). "Uncertainty Management for Engineering Systems Planning and Research". In: *Engineering Systems Monographof the Engineering Systems Symposium*. MIT, Cambridge, MA.

De Weck, O. (2004). "Multi-Objective Optimization: History and Promise". Keynote Paper. The 3rd China-Japan-Korea Joint Symposium on Optimization of Structural and Mechanical Systems, Kanazawa, Japan.

De Weck, O. and Willcox, K. (2005). *Lecture Notes for 16.888 Multidisciplinary System Design Optimization course notes*. MIT, Cambridge, MA.

Deb, K. (2001). *Multi-Objective Optimization Using Evolutionary Algorithms*. Chichester: John Wiley & Sons.

DeMarco, T. (1979). *Structured Analysis and System Specification*. Yourdon Press Computing Series. Eaglewood Cliffs, New Jersey.

Dieter, G. (2000). *Engineering Design-A Material and Processing Approach*. McGraw-Hill, Boston, Massachusetts.

Dietrich, W.C. (1989). "Saving a Legacy with Objects," In *Proceedings of OOPSLA-90*, Addison-Wesley, Reading, MA.

Dixon, J. and Poli, C. (1995). *Engineering Design and Design for Manufacturing, A Structured Approach*. Field Stone Publishers. Conway, Massachusetts.

Dixon, J.R. (1987). *On Research Methodology towards a scientific theory of engineering design*. AI EDAM. 1 (3), pp. 145-157.

Downing, F. and Flemming, U. (1981). "The Bungalows of Buffalo". *Environment and Planning* B. 8(3), pp. 269 – 293.

Duarte, J. (2001). "Customizing Mass Housing: a Discursive Grammar for Siza's Malagueira houses". Ph.D dissertation. Massachusetts Institute of Technology. Cambridge, Massachusetts.

Dumitrescu, D., Lazzerini, B., Jain, L. and Dumitrescu, A. (2000). *Evolutionary Computation*. Boca Raton, FL: CRC Press.

Duvvuru, S., Stephanopouls, G. Logcher, R., et al. (1989). "Knowledge-based system applications in engineering design: Research at MIT". *AI Magazine*. 10(3), pp. 79-96.

Eastman C., Teicholz, P., Sacks, R. and Liston K. (2008). *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*. Wiley Publishing. Hoboken, New Jersey.

Edgeworth, F.Y. (1881). *Mathematical Psychics,* P. Keagan, London, England.

Eggert, R.J. (2004). *Engineering Design*. Prentice Hall, Upper Saddle River, New Jersey.

Elster, K. (1993). *Modern Mathematical Methods of Optimization*. Wiley VCH. Berlin, Germany.

Eppinger, S. (1997). "A Planning Method for Integration of Large-Scale Engineering Systems". *Inte. Conference on Engineering Design ICED-97*. Tampere, Finland, pp. 199–204.

Eppinger, S. (1991). "Model--based Approaches to Managing Concurrent Engineering". *Journal of Engineering Design*. 2(4),pp. 283-290.

Eppinger, S. D. and Gebala, D. A. (*1991*) "Methods for Analyzing Design Procedures". *ASME Conference on Design Theory and Methodology. Miami, FL.* pp. 227-233.

Eppinger, S. Nukala, M. and Whitney, D. (1997). "Generalized Models of Design Iteration Using Signal Flow Graphs". *Research in Engineering Design*. 9, pp.112-123.

Fenves, S. and Baker, N. (1987). “Spatial and functional representation language for structural design”. In:Gero, J. (Ed.). *Expert Systems in Computer-Aided Design*. Elsevier Science, North-Holland, pp. 511-529.

Finger, S. and Rinderle, J. R. (1989). “A Transformational Approach to Mechanical Design Using a Bond Graph Grammar. Proceedings of the First ASME Design Theory and Methodology Conference.

Fonseca C. and Fleming P. (1993) “Genetic Algorithms for Multi-objective Optimization: Formulation, Discussion and Generalization”. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*. pp 416-423, San Mateo, California. University of Illinois at Urbana-Champaign, Morgan Kaufman Publishers

Gallopoulos, E. Houstis, E. and Rice, J. (1994). “Computer as Thinker/Doer: Problem-Solving Environments for Computational Science”. *Computing in Science and Engineering*. 1(2), pp. 11-23.

Gao, J. Tsao, S. and Wu, Y. (2003). *Testing and Quality Assurance for Component-based Software*. Artech House, Norwood, Massachusetts.

Gero, J. S. (1990). “Design Prototypes: a Knowledge Representation Schema for Design”. *AIMagazine*.11 (4): 26-36.

Gershenfeld, N. (1998). *Nature of Mathematical Modeling*. Cambridge University Press. New York, New York.

Gershenson, J. K., Prasad, G. J. and Zhang, Y. (2003). “Product modularity: definitions and benefits”. *Journal of Engineering Design*, 14:3, 295 — 313

Glover, F. (1990). “Tabu Search - Part 2”. *ORSA Journal on Computing*. 2(1), pp. 4-32.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Publishing Company. Reading, Massachusetts.

Grady, J. (1994). *System Integration*. CRC Press. Ann Arbor, Michigan.

Graham, B. (2004). *Detail Process Charting: Speaking the Language of Process*. Wiley. Hoboken, New Jersey.

Granville, V. Krivanek, M. and Rasson, J. (1994). *“Simulated Annealing: A Proof of Convergence”.* IEEE Transactions on Pattern Analysis and Machine Intelligence. *16 (6), pp. 652-656.*

Gries, M. (2004). “Methods for evaluating and covering the design space during early design development”. Integration the VLSI Journal.38 (2), pp. 131-183.

Harrison, G. Maynard, D. and Pollak, E. (2004). "Automated Database and Schema-based Data Interchange for Modeling and Simulation". Proceedings of the 2004 Winter Simulation Conference, pp. 191–197.

Hastings, D. (2004). Lecture Notes for 16.892J / ESD.353J Space System Architecture and Design. MIT, Cambridge, MA.

Hayes, B. (2000). "Graph Theory in Practice: Part I". *American Scientist*. 88(1), pp. 9–13.

Hedberg, S.R. (2005). "Evolutionary Computing: the Rise of Electronic Ereeding". *Intelligent Systems*, IEEE. 20(6), pp. 12 – 15

Heisserman, J. Callahan, S. and Mattikalli, R. (2000). "A design representation to support automated design generation". In: Gero J. (ed). *Artificial Intelligence in Design*. pp 545-566.

Helton, J.C., Johnson, J.D. Salaberry, C.J. and Storlie, C.B. (2006). "Survey of Sampling Based Methods for Uncertainty and Sensitivity Analysis". *Reliability Engineering and System Safety*. 91, pp.1175–1209.

Hemberg, M. (2001). Genr8 - a design tool for surface generation. Master's thesis, Chalmers University of Technology, Goteborg, Sweden.

Hertz, A. Taillard, E. and de Werra, D. (1997). "Tabu Search". In: Aarts, E. and Lenstra, J. (Eds.). *Local Search in Combinatorial Optimization*. John Wiley &Sons, pp.121-136.

Holland, J. (1992) *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press. Cambridge, Massachusetts.

Hongo, K. (1985). "On the significance of the theory of design". In: Yoshikawa, H. (Ed.). *Design and Synthesis*. Elsevier Science Publishers. Amsterdam.

Hornby, G. S., and Pollack, J. B. (2001a). "The Advantages of Generative Grammatical Encodings for Physical Design. Proceedings of the 2002 Congress on Evolutionary Computation.

Hubka, V., Andreasen, M. and Eder, W. (1988) *Practical Studies in Systematic Design*. Butterworths. London.

Huebner, K. Dewhirst, D. Smith, D. Byrom, T. (2001). *The Finite Element Method for Engineers*. Wiley-IEEE. Hoboken, New Jersey.

INCOSE (2002). *Systems Engineering Handbook: A "How To" Guide for All Engineers*, 2nd ed., International Council on Systems Engineering, Seattle, WA, pp. 19–28.

Jacoby, S. and Kowalik, J. (1980). *Mathematical Modeling with Computers*. Prentice-Hall, Englewood Cliffs, New Jersey.

Jaki S. L. (1981). *Immanuel Kant: Universal Natural History and the Theory of the Heavens*. Scottish Academic Press.

Jo, J. and Gero, *J. S. (*1998). "Space layout planning using an evolutionary approach". *Artificial Intelligence in Engineering*. 12(3): 149-162

Kalay, Y. (1989). *Modeling Objects and Environments*. Wiley Publishing. Hoboken, New Jersey.

Kalay, Y. (2004). *Architecture's New Media: Principles, Theories, and Methods of Computer-Aided Design*. The MIT Press. Cambridge, Massachusetts.

Kalyanmoy Deb and David E. Goldberg. (1989). "An Investigation of Niche and Species Formation in Genetic Function Optimization". In J. David Schafer, editor, Proceedings of the Third International Conference on Genetic Algorithms, pages 42-50, San Mateo, California, June. George Mason University, Morgan Kaufmann Publishers.

Kaymaz, I. (2005). "Application of Kriging Method to Structural Reliability Problems". *Structural Safety*. 27 (2), pp.133–151.

Keskin, A. (2007). *Process Integration and Automated Multi-Objective Optimization Supporting Aerodynamic Compressor Design.* Ph.D. Dissertation. Brandenburg University, Berlin.

Khedro, T. (1996). "A distributed problem-solving approach to collaborative facility engineering". *Advances in Engineering Software***.** 25(2-3), pp. 243-252.

Kirkpatrick, S. Gelatt, C. and Vecchi, M. (1983). "Optimization by Simulated Annealing". Science. New Series 220 (4598), pp. 671-680.

Koch, P. Evans, J. and Powell, D. (2002). "Interdigitation for effective design space exploration using iSIGHT". *Structural and Multidisciplinary Optimization*. 23(2), 111–126.

Kockler, F. Withers, T. Poodiack, J. and Gierman, M. (1990). *Systems engineering management guide.* Defense Systems Management College. U.S. Government Printing Office. Washington, D.C.

Koning, H. Eizenberg, J. (*1981*). "The Language of the *Prairie*: Frank Lloyd *Wright's Prairie houses*". *Environment and Planning* B. 8(3), pp. 295 – 323.

Kossiakoff, A. and Sweet, W. (2002). *Systems Engineering Principles and Practice*. Wiley-Interscience, Hoboken, New Jersey.

Kroo, I.M. (1997a). "MDO for large-scale design". In: Alexandrov, N. M. and Hussaini M. Y. (Eds.), *Multidisciplinary Design Optimization: State of the Art*. SIAM, 1997, pp. 22-44.

Kroo, I.M. (1997b). "Multidisciplinary optimization applications in preliminary design – Status and Directions". 38th AIAA/ASME/ASCE/AHS/ASC, Structures, Structural Dynamics and Materials Conference, Kissimmee, FL.

Kuhn, T.S. (1970). *The Structure of Scientific Revolutions*. University of Chicago Press. Chicago, IL.

Law, A. and Kelton, W. D. (1999). *Simulation Modeling and Analysis*. McGraw-Hill, New York.

Lawson, B. (2005). *How Designers Think: The Design Process Demystified*. Architectural Press-Elsevier. Boston, Massachusetts.

Li, M. Rana, O. Walker, D. Shields, M. and Huang, Y. (2004). "Component-based Problem Solving Environments for Computational Science". In: Lau, K. (Ed.). *Component-based Software Development: Case Studies*. World Scientific. New Jersey.

Liebeck, R., Page, M. and Rawdon, B. (1996) "Evolution of the Revolutionary Blended Wing Body Subsonic Transport". *Transportation Beyond 2000: Technologies Needed for Engineering Design*, NASA TP 10184, pp. 418-459.

Littlejohn, S.W. (1998). *Theories of Human Communication*. Wadsworth Publishing Company. Belmont, California.

Long, J. (2002). *Relationships between Common Graphical Representations in System Engineering*, Vi-Tech Corporation. Vienna, Virginia.

Luenberger, D. (2003). *Linear and Nonlinear Programming*. Springer, New York, New York.

Magee, C., Moses, J. and Whitney, D. (2006). Lecture Notes for ESD.342 Advanced System Architecture. MIT, Cambridge, MA.

Maki, D. and Thompson, M.(2006). *Mathematical Modeling and Computer Simulation*. Thomson Brooks/Cole. Belmont, California.

Malone, T. and Crowston, K. (1991). "Towards an interdisciplinary theory of coordination". *MIT Sloan School Working*, Paper 3294-91-MSA. Massachusetts Institute of Technology, Cambridge, MA.

Mandelbrot, B. (1983). *The Fractal Geometry of Nature*. W. H. Freeman. New York, New York.

March, L. (1976). "The Logic of Design and the Question of Value". In: March, L. (Ed.) *The Architecture of Form*. Cambridge University Press. Cambridge.

Markus, T.A. (1969). “The Role of Building Performance Measurement and Appraisal in Design Method”. In: Broadbent, G and Ward, A. (Eds.). *Design Methods in Architecture*. George Wittenborn, New York, New York.

Maver, T. (1970). *Appraisal in the Building Design Process: Emerging Methods in Environmental Design and Planning*. MIT Press. Cambridge, Massachusetts.

McCord, K. (1993). “Managing the Integration Problem in Concurrent Engineering”. Master of Science Thesis. Massachusetts Institute of Technology. Cambridge, Massachusetts.

McCormack, J. and A. Dorin (2001). “Art, Emergence, and the Computational Sublime”. *Second Iteration: Proceedings of the Second International Conference on Generative Systems in the Electronic Arts*, Melbourne.

McKay, M., Bechman R. and Conover, W. (1979). “A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code”. *Technometrics*. 21(2), pp. 239-245.

McManus, H. Hastings, and D. Warmkessel, J. (2004). *Journal of Spacecraft and Rockets* 41(1), pp. 10-19.

McManus, H., Hastings, D. and Warmkessel, J. (2004). “New Methods for Rapid Architecture Selection and Conceptual Design”. *Journal of Spacecraft and Rockets*, Vol. 41, No.1, Jan.-Feb., pp. 10-19.

Meredith, D.D., Wong, K.W., Woodhead, R.W. and Wortman, R.H. (1985). *Design and Planning of Engineering Systems*. Prentice-Hall, Englewood Cliffs, New Jersey.

Messac, A. (2000). “From Dubious Construction of Objective Functions to the Application of Physical Programming”. *AIAA Journal*, 38(1), pp. 155-163.

Messerschmitt, D. and Szyperski, C. (2003). *Software Ecosystem – Understanding An Indispensable Technology and Industry.* MIT Press, Cambridge, Massachusetts.

Miller, G.A. (1956). “The Magical Number Seven, Plus or Minus Two”. *The Psychological Review*. 63(2), pp. 81-97.

Minsky, M. L. (1988). *The Society of Mind*. Simon and Schuster. New York, NY.

Mitchell, W. (1986). “Formal representations: a foundation for computer-aided architectural design”. *Environment and Planning* B. 13, pp.133-162.

Mitchell, W. (1990). *The Logic of Architecture: Design, Computation, and Cognition*. MIT Press. Cambridge, MA.

Mitchell, W. (1991). "Functional Grammars: An Introduction". In: Goldman, G. and Zdepski, M. (eds.) *Reality,and Virtual Reality* (ACADIA 1991).pp. 167-176. Troy, New York.

Mowbray, T. and R. Zahavi (1994), *The Essential CORBA*, Wiley, New York.

Mullens, M., Arif, M., Armacost, R., Gawlik, T. and Hoekstra, R. (2005). "Axiomatic Based Decomposition for Conceptual Product Design". *Production and Operations Management*.14 (3), pp. 286–300.

Myers, R. and Montgomery, D. (1995). *Response surface methodology: process and product optimization using designed experiments*. John Wiley & Sons. New York, New York.

National Research Council (2006). *Network Science, Committee on Network Science for Future Army Applications*. The National Academies Press. Washington D.C.

Newman, M. (2003). *The Structure and Function of Complex Networks*. SIAM Review45, pp. 167–256.

NIST. (2006). *Engineering Statistics Handbook*. SEMATECH.

Nocedal, J. and Wright, S. (2000). *Numerical Optimization*. Springer. New York, New York.

Object Management Group. (2008). "CORBA Basics". Retrieved August 15, 2008. Website: http://www.omg.org/gettingstarted/corbafaq.htm

Object web open source Middleware. (2008). "What is Middleware". Retrieved September 5, 2008. Website: http://middleware.objectweb.org/

Oliver, D. (1994). "Systems Engineering and Object Technology". *Proceedings of the 4th International Council on Systems Engineering International Symposium*, San Jose, p. 315.

OMG (2007a) *OMG Systems Modeling Language* (OMG SysML™) website: *http://www.omgsysml.org/*.

OMG (2007b) *OMG Systems Modeling Language* (OMG SysML™) Specification. SysML 1.0 Proposed Available Specification (PAS), OMG document [ptc/2007-02-03] dated 2007-03-23.

Pahl, G. and Beitz, W. (1988). *Engineering Design*. Springer-Verlag. Berlin.

Pahng, F., Senin, N. and Wallace, D. (1997). "Modeling and evaluation of product design problems in a distributed design environment". Proceedings of the 1997 ASME Design Engineering Technical Conferences, Sacramento, California, September 14–17, DETC97/DFM-4356, CD-ROM (New York: ASME).

Papadimitriou, C. and Steiglitz, K. (1998). *Combinatorial optimization: Algorithms and Complexity*. Courier Dover Publications. New York, New York.

Papalambros, P. (2000). “Extending the Optimization Paradigm in Engineering Design”. Proceedings of the 3rd International Symposium on Tools and Methods of Competitive Engineering, Delft, The Netherlands.

Papalambros, P. and Wilde, D. (2000). *Principles of Optimal Design*. Cambridge University Press. New York, New York.

Pareto, V. (1906). *Manuale di Economia Politica*, Societa Editrice Libraria, Milano, Italy. Translated into English by A.S. Schwier, (1971). As *Manual of Political Economy*, Macmillan, New York.

Parnas, D. (1972). “Information Distribution Aspects of Design Methodology”. *Proceedings of the 1971 IFIP Congress*. Ljubljana, Yugoslavia, August 23-28, pp. 339-344. Amsterdam, Netherlands: North-Holland Publishing Company.

Parnas, D.L.: “On the Criteria To Be Used in Decomposing Systems Into Modules”. *Communications of the ACM*. 15,12. (1972) 1053-1058

Paydarfar, S. (2001). “An Integration Maturity Model for the Digital Enterprise”. *The Digital Enterprise*. pp. 29-44.

Pender T. A. (2002). *UML Weekend Crash Course*. Wiley Publishing. Hoboken, New Jersey.

Perry, D.E. and Wolf, A.L. (1992). *Foundations for the Study of Software Architecture*. ACM SIGSOFT. 17(4), pp. 40-52.

Phoenix Integration, Inc. (2007). ModelCenter 7.0 Help Documentation.

Phoenix Integration. (2004). “Design Exploration and Optimization Solutions”. Technical White Paper.

Pimmler, T. and Eppinger, S. (1994). “Integration Analysis of Product Decompositions”. Proceedings of ASME Design Theory and Methodology Conference. DE- 68, pp. 343-51.

Prusinkiewics, P. and Lindenmayer, A. (1991) *The Algorithmic Beauty of Plants*. Springer-Verlag.

Pugh, S. and Morley, I. (1989). “Organizing for Design in Relation to Dynamic/Static Product Concepts”. Proc. Sixth Int. Conf. on Engineering Design. Harrogate, UK, 1, pp.313-334.

Rao, S. (1996). *Engineering Optimization: Theory and Practice*. Wiley-IEEE. Hoboken, New Jersey.

Requicha, A. (1980). "Representations of Rigid Solids: Theory, Methods and Systems". *ACM Computing Surveys*. 12(4), pp. 437-466.

Reynolds, F. Natrajan, A., Srinivasan S. (1997). "Consistency maintenance in multiresolution simulation". *ACM Transactions on Modeling and Simulation*, 7(3) pp. 368-392.

Ridge, E. (2007). *Design of Experiments for the Tuning of Optimization Algorithms*, PhD Thesis. The University of York.

Rinderle, J. (1991). "Grammatical Approaches to Engineering Design, Part II: Melding Configuration and Parametric Design Using Attribute Grammars". *Research in Engineering Design. 2*, 137-146.

Rosenman, M. and Simoff, S. (2001). "Some conceptual issues in component-assembly modeling". *Artificial Intelligence in Engineering*. 15(2), pp. 109-119.

Rowe, P. G. (1987). *Design Thinking*. MIT Press. Cambridge, Massachusetts.

Roy, R. (2001). *Design of Experiments Using The Taguchi Approach: 16 Steps to Product and Process Improvement*. Wiley-Interscience. New York

Royal Institute of British Architects *(1965). Architectural Practice and Management Handbook.*

Sacks, J. Welch, W., Mitchell, T. and Wynn, H. (1989). "Design and Analysis of Computer Experiment". *Statistical Science*. 4, pp. 409-435.

Sacks, R. Eastman, C. Lee, G. (2004). "Parametric 3D Modeling in Building Construction with Examples from Precast Concrete". *Automation in Construction*. 13, pp 291– 312.

Sage, A.P. and Armstrong Jr., J.E. (2000). *Introduction to Systems Engineering*. Wiley-Interscience. Malden, Massachusetts.

Sakalkar, V. and Hajela, P. (2008). "Multilevel Decomposition Based Non-Deterministic Design Optimization". 49th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference. Schaumburg, IL.

Sakata, A. Ashida, F. and Zako, M. (2003). "Structural Optimization Using Kriging Approximation". *Computer Methods in Applied Mechanics and Engineering*.192 (7–8), pp. 923–939.

Savic, D. (2002). "Single-objective *vs.* Multi-Objective optimization for integrated decision support". In: Rizzoli A. and Jakeman A. J. (Eds.). *Integrated Assessment and Decision Support* . Proceeding of First Biennial Meeting of the Int. Environmental Modeling and Software Society, 1, 7–12.

Schmidt, D. (2001). “Object Interconnections: CORBA and XML - Part 3: SOAP and Web Services”. Retrieved August 14,2008. Website: http://www.ddj.com/cpp/184403802

Schmidt, J.W. and Taylor, R.E. (1970). *Simulation and Analysis of Industrial Systems*. Richard D. Irwin, Homewood, Illinois.

Scholz-Reiter, B. and Stickel, E. (Eds.). (1996) *Business Process Modelling*. Springer-Verlag. Berlin.

Seacord, R. Comella-Dorda, S. Lewis, G. Place, P. and Plakosh, D.(2001). *Legacy System Modernization Strategies*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University.

Senin N, Wallace D, Borland N. (1999). “Object-based design modeling and optimization with genetic algorithms”. GECCO-99: Proceedings of the genetic and evolutionary computation conference, 13–17 July, Orlando, FL.

Shah, J. and MŠntylŠ, M. (1995).*Parametric and Feature-Based CAD/CAM: Concepts, Techniques, and Applications*. Wiley Publishing. Hoboken, New Jersey.

Shea, K. and Luebkeman, C. (2005). “CDO: Computational Design + Optimization in Building Practice”. In: *The Arup Journal*, 3.

Shea, K., Aish, R., and Gourtovaia, M. (2005). “Towards Integrated Performance-driven Generative Design Tools”. *Automation in Construction*. 14(2), pp. 253-264.

Shi, Q. Hagiwara, I. and Takashima, F. (1999) “The Most Probable Optimal Design Method for Global Optimization”.In: Proceedings of the 1999 ASME Design Engineering Technical Conferences. Las Vegas, Nevada, pp. 12–15.

Shields, M. Rana, O. Walker, D. Li, M. and Golby, D. (2000) “A Java/CORBA Based Visual Program Composition Environment for PSEs”. *Concurr.: Pract. Exp.* **12(**8), pp. 687-704.

Simon, H. A. (1973). *The Sciences of the Artificial*. MIT Press. Cambridge, MA.

Simpson, T.W.; Peplinski, J.; Koch, P N.; Allen, J.K. (1997): “On the Use of Statistics in Design and the Implications for Deterministic Computer Experiments”. *Design Theory*& *Method* – DTM’97 (held in Sacramento, CA), ASME Paper No. DETC97/DTM-3881

Smith R., Eppinger S. (1997). “Identifying Controlling Features of Engineering Design Iteration”. *Management Science*. 43(3), March, pp. 276-293.

Smith, R. Eppinger, S. (1996). “A Predictive Model of Sequential Iteration in Engineering Design”. Management Science.

Sneed, H. (2000). "Encapsulation of legacy software: A technique for reusing legacy software components". *Annals of Software Engineering*. 9, pp. 293–313.

Software Engineering Institute. Carnegie Mellon. (2008). "Software System Integration". Retrieved September 2, 2008. Website: http://www.sei.cmu.edu/productlines/frame_report/softwareSI.htm

Steadman, P. (1979) *The Evolution of Designs*. Cambridge University Press. Cambridge.

Sterman, J. (2000). *Business Dynamics: Systems Thinking and Modeling for a Complex World*. McGraw-Hill, Boston.

Steward, D.V. (1981). "The Design Structure System: A Method for Managing the Design of Complex Systems, IEEE Trans Eng Management". *EM*-28(3), pp. 71–74.

Stiny, G. (1982). "Spatial relations and grammars". *Environment and Planning* B 9, 313–314.

Stiny, G. and Gips, J. (1972). *Shape Grammars and the Generative Specification of Painting and Sculpture*. C V Freiman (Ed) Information Processing 7. Amsterdam, North–Holland, 1460–1465.

Stiny, G. and Mitchell, W. J. (1978) "The Palladian Grammar".*Environment and Planning* B 5: 5-18.

Strogatz, S. (2001). "Exploring Complex Networks". *Nature*410 (8 March), pp. 268-276.

Suh, N. Bell, A. and Gossard, D. (1978) "On an Axiomatic Approach to Manufacturing and Manufacturing Systems". *ASME Journal of Engineering for Industry*. 100, pp. 127-130.

Suh, N.P. (1990). *The Principles of Design*. Oxford University Press. New York.

Sun, X. and Blatecky, A. (2004). "Middleware: The Key to Next Generation Computing". Preface for the *Journal of Parallel and Distributed Computing*, pp. 689–691.

Sydenham, P. (2003). *Systems Approach to Engineering Design*. Artech House Publishers.

Terzidis, K. (2006). *Algorithmic Architecture*. Architectural Press, Elsevier. New York, New York.

Tong, S. S. (2001). "The Software Robot: A New Paradigm in Computational Engineering". *Proceedings of the Conference on Computational Engineering and Science*. 6(1), pp.1-4.

Trefethen, L. and Bau, D. (1997). *Numerical Linear Algebra*. SIAM, Philadelphia, Pennsylvania.

Tyng, A. (1984) *Beginnings*. Wiley. New York.

Ulrich, K. (1995). "The Role of Product Architecture in the Manufacturing Firm". *Research Policy*. 24(3), pp. 419-440.

Ulrich, K. and Seering, W.P. (1990). "Function Sharing in Mechanical Design". *Design Studies*. 11(4), pp. 223-234.

Ulrich, K.T. and Ellison, D. J. (1999) "Holistic Customer Requirements and the Design-Select Decision". *Management Science*. 45(5), pp. 641-658.

Ulrich, K.T., and Eppinger, S.D. (2000). *Product Design and Development*. McGraw-Hill. New York. New York.

Van Laarhoven, P. and Aarts, E. (1987). *Simulated Annealing: Theory and Applications*. D. Reidel Publishing Company.

VDI 2221 (1985). *Systematic Engineering Design of Technical Systems and Products* .VDI Richtlinien, Verein Deutscher Ingenieure.

Wallace, D. Abrahamson, S. Senin, N. and Sferro, P. (2000). "Integrated Design in a Service Marketplace". *Computer-aided Design*. 32(2), pp.97-107.

Ward, P. and Mellor, S. (1985). *Structured Development for Real Time Systems*. Vols 1-3. Yourdon Press Computing Series. Eaglewood Cliffs, New Jersey.

Watts, D. J., and S. Strogatz. (1998). "Collective Dynamics of 'Small-World' Networks". *Nature.*393 (4 June), pp. 440-442.

Whitney, D.E. (1996). "Why Mechanical Design Cannot Be Like VLSI Design". *Research in Engineering Design*.8 (3), pp. 125- 138.

Wirth, N. (1976). *Algorithms + Data Structures = Programs. Prentice-Hall.*

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media. Champaign, *Illinois.*

Wolsey, L. and Nemhauser, G. (1999). *Integer and Combinatorial Optimization*. Wiley-Interscience. Hoboken, New Jersey.

Yessios, C. (1975). "Formal Languages for Site Planning". In: Eastman, C.M. (Ed.). *Spatial Synthesis in Computer-Aided Buildings Design*. Wiley, New York, New York.

Yilmaz, L. and Ören, T.I. (2004). "Dynamic Model Updating in Simulation with Multi-models: A Taxonomy and a Generic Agent-Based Architecture". *Proceedings of SCSC 2004 - Summer Computer Simulation Conference, San Jose, CA*. pp. 3-8.

Zachman, J.A. (1987). "A framework for information systems architecture". *IBM Systems Journal*. 26(3), pp. 276-292.

Zeigler, B. P., Praehofer, H. Kim, G. T. (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press.