A Meta-language for Systems Architecting

by

Hsueh-Yung Benjamin Koo

M.S. in Engineering Systems and Management

Massachusetts Institute of Technology, 2001

Submitted to the Engineering Systems Division
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2005

© 2005 Hsueh-Yung Benjamin Koo. All rights reserved

Author……………………………………………………………………………………….................
Engineering Systems Division
January 31, 2005

Certified by…………………………………………………………………………………..........................
Edward F. Crawley
Professor of Aeronautics and Astronautics and Engineering Systems
Thesis Supervisor

Certified by…………………………………………………………………………………..........................
Christopher Magee
Professor of the Practice of Engineering Systems and Mechanical Engineering
Thesis Advisor

Certified by…………………………………………………………………………………..........................
Dov Dori
Associate Professor of Industrial Engineering, Technion, Israel Institute of Technology
Thesis Advisor

Certified by…………………………………………………………………………………..........................
Olivier L. de Weck
Assistant Professor of Aeronautics and Astronautics and Engineering Systems
Thesis Advisor

Accepted by…………………………………………………………………………………..........................
Richard de Neufville
Professor of Civil and Environmental Engineering and Engineering Systems
Chair, Engineering Systems Division Education Committee

道可道非常道　名可名非常名
無名天地之始　有名萬物之母
故常無　欲以觀其妙
常有　　欲以觀其徼
此兩者　同出而異名　同謂之玄
玄之又玄　衆妙之門

老子

A Meta-language for Systems Architecting

by
Hsueh-Yung Benjamin Koo

Submitted to the Engineering Systems Division
on January 31, 2005, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The aim of this research is to design an executable meta-language that supports system architects' modeling process by automating certain model construction, manipulation and simulation tasks. This language specifically addresses the needs in systematically communicating architects' intent with a wide range of stakeholders and to organize knowledge from various domains. Our investigation into existing architecting approaches and technologies has pointed out the need to develop a simple and intuitive, yet formal language, that expresses multiple layers of abstractions, provides reflexive knowledge about the models, mechanizes data exchange and manipulation, while allowing integration with legacy infrastructures. A small set of linguistic primitives, stateful objects and processes that transform them were identified as both required and sufficient building blocks of the meta-language, specified as an Object-Process Network (OPN). To demonstrate the applicability of OPN, a software environment has been developed and applied to define meta-models of large-scale complex system architectures such as space transportation systems.

OPN provides three supporting aspects of architectural modeling. As a declarative language, OPN provides a diagrammatic formal language to help architects specify the space of architectural options. As an imperative language, OPN automates the process of creating architectural option instances and computes associated performance metrics for those instances. As a simulation language, OPN uses a function-algebraic model to subsume and compose discrete, continuous, and probabilistic events within one unified execution engine.

To demonstrate its practical value in large-scale engineering systems, the research applied OPN to two space exploration programs and one aircraft design problem. In our experiments, OPN was able to significantly change the modeling and architectural reasoning process by automating a number of manual model construction, manipulation, and simulation tasks.

Thesis Supervisor: Edward Crawley

Thesis Advisor: Christopher Magee

Thesis Advisor: Dov Dori

Thesis Advisor: Olivier de Weck

Acknowledgement

My deepest gratitude goes to Prof. Edward Crawley whose inexhaustible ideas and energy fueled the development of this thesis. He also taught me how to think on my feet and survive in this fertile and often overwhelming research area. The concepts of an abstract and flexible meta-language and executable software tool as described in this thesis would not have taken a concrete form without his unwavering support in this extremely risky project. Due to his contribution, members of our research team have often attached his name to the executable meta-language: Crawley Machine.

Prof. Dov Dori, inventor of Object Process Methodology (OPM), whose pioneering work on a domain-neutral bi-modal language framework paved the road for my research. He was also my masters' thesis advisor, which eventually led to the development of my research work in the doctoral program. Prof. Dori has not only served as a thesis advisor, he has also created many important career development opportunities for me. I am extremely grateful for his support in work and in life.

Prof. Chris Magee led me to demonstrate the practical value of my research. His wisdom helped me see the philosophical foundation of my research in a different light. His vision into the practical use of my work helped me develop a more intuitive articulation of architectural reasoning.

Thanks also go to Prof. Daniel Hastings, Prof. Paul Carlile, and Prof. Olivier de Weck. These professors went out of their ways to get me admitted as one of the students in the pilot doctoral program at the Engineering Systems Division. During my doctoral studies, they actively formulated many educational opportunities that are particularly beneficial to my intellectual development.

The experimental knowledge about two of the case studies are provided by Dr. Robert C. Seamans and Mr. Willard Simmons. Dr. Seamans went through the historical account of the Apollo Program and gave me access to valuable documents that would not be easily attainable. Mr. Simmons applied the untested software tool and helped me develop the computational techniques to better demonstrate the practical value of my work. Mr. Christopher Fry's work on the first software prototype gave me significant support in developing the technologies that ultimately lead to the development of the software tool documented in this thesis.

Mr. Jay Conne and Dr. Geilson Lorero offered epistemological insights and expertise in system engineering to better organize my thesis. Their ongoing efforts to identify potential applications of OPN inspired my further understanding of the issues and its potential for practical uses.

Many people reviewed the drafts of my thesis. They provided many useful and insightful comments. They are David Loda, Dr. Roger Chang, Jason Cawley, Ryan Boas, Matt Silver, Elim Qiu, Russ Wertenberg, Annie-Pierre Hurd, Charlie Stromeyer, Daniel Krech, and Chun-Ming Yang.

I am also grateful to many friends who devoted their personal time to help me on both intellectual and personal levels. It is their support that got me through the hurdles during the years in my doctoral study. Beside the names mentioned above, they include: Liu Chih-Hung, Peter Panetta, Prof. Zhong Xia Liang, Karen Marais, Rudolf Smaling, Prof. James Hines, James Rice, Dr. J. C. Duh, Kathi Grace, Yuan Gao, Dr. Daniel Whitney, Prof. Thomas Malone, Dr. David Willmes, Keen Sing Lee, Wilfried Hofstetter, Brennan McCarragher, William Litant, Dr. Stephan Wolfram, Prof. Nam Suh, Thomas Coffee, Prof. Joel Cutcher-Gershenfeld, Prof. Joel Moses, Prof. David Mindell, James Cheng, Prof. James Utterback, Dr. Robert Bayt, Dr. Pengju Kang, Col. Peter Young, Victor Tang, Chris Anderson, Dr. Abbott Weiss, James Tsao, Dr. Ernst Fricke, Prof. Dennis Mahoney, Joost Bonsen, Lois Slavin, Prof. Paul Lagace, and Prof. Pat Hale.

During the last year of my doctoral program, I met and married Michi Wang. She provides a great source of joy and a sense of direction in my life. For that, I am forever grateful. I also want to thank my parents, Tawo Koo and Yuan Chen Yen, for their loving support. My father's interests in linguistics seeded my intellectual roots. My mother's creativity and perseverance inspired me to see opportunities in every difficult situation. Without their dedication and patience toward me, this work would not have begun.

Financial assistance provided by: the Charles Stark Draper Laboratory, Inc. and NASA.

TABLE OF CONTENTS

1 INTRODUCTION **8**

1.1 BACKGROUND 9
1.2 AIM AND OBJECTIVES 13
1.3 RESEARCH APPROACH 13
1.3.1 Theory *13*
1.3.2 Tool *14*
1.3.3 Application *15*
1.4 THESIS SYNOPSIS 15

2 LITERATURE REVIEW **17**

2.1.1 Theoretical Foundation *17*
2.1.2 Qualitative Methods *20*
2.1.3 Quantitative Methods *23*
2.2 LANGUAGES FOR ARCHITECTING 26
2.2.1 Pattern languages *26*
2.2.2 System Description Languages *27*
2.2.3 Generative Modeling Techniques *34*
2.2.4 Simulation Languages *38*
2.2.5 Meta-languages *48*
2.3 SUMMARY OF REVIEWED MODELING LANGUAGES 52
2.3.1 Comparative Studies of modeling languages *52*
2.3.2 Emphasis of modeling languages *53*

3 NEEDS AND REQUIREMENTS **56**

3.1 THE NEEDS OF SYSTEMS ARCHITECTING 56
3.2 THREE TYPES OF ARCHITECTURAL REASONING TASKS 57
3.3 REQUIREMENTS OF THE ARCHITECTS' META-LANGUAGE 59
3.3.1 Subsume various models of computation *59*
3.3.2 Generate possible subsets of alternatives within finite time *60*
3.3.3 Adaptively construct computable expressions *60*
3.3.4 Enable Model Introspection *61*
3.3.5 Support layered abstraction models *62*
3.3.6 Diagrammatically represent system models *62*
3.3.7 Deploy across standard computing platforms *63*
3.4 A SOLUTION PROFILE 64

4 AN EXECUTABLE META-LANGUAGE: OBJECT-PROCESS NETWORK **65**

4.1 THE SPACE OF MODELING LANGUAGES 65
4.2 THING, RELATIONSHIP AND GRAPH 66
4.3 OPERANDS AND OPERATORS 67
4.3.1 The meta-operand: Thing *68*
4.3.2 The meta-operator: Eval *70*
4.3.3 Notations *71*
4.4 SYNTAX 74
4.5 SEMANTICS 76
4.5.1 A layered semantic model *77*
4.6 TOKEN GENERATION AND SCHEDULING 77
4.6.1 The execution model of Eval *78*
4.6.2 Turing Completeness *81*
4.6.3 Model Enumeration in Finite Time *82*

4.7 TOKEN PROCESSING 86
4.8 VARIABLE BINDING 91

5 THE SOFTWARE ENGINEERING ASPECTS OF OPN (PRAGMATICS) 93

5.1 IMPLEMENTATION OBJECTIVES 93
5.2 THE DESIGN OF THE LANGUAGE KERNEL 94
5.2.1 Model 95
5.2.2 View 95
5.2.3 Controller 95
5.3 USER INTERFACE DESIGN 96
5.3.1 Visualizing computationally generated OPN models 97
5.4 USER INTERFACE FRAMEWORK FOR LAYERED SEMANTICS 98
5.4.1 User Interface for Token Generating and Scheduling 98
5.4.2 User Interface for Token Processing 100
5.4.3 User interface for Variable Binding 101
5.4.4 User Interface for model detail inspection 102
5.5 ENABLING TECHNOLOGIES 105

6 CASE STUDIES 109

6.1 THE APOLLO PROGRAM (RETROSPECTIVE) 109
6.1.1 Where an executable meta-language is applicable 110
6.1.2 Specifying mission architectures in formal languages 113
6.1.3 Generate all possible trajectories 119
6.1.4 Calculating performance metrics 123
6.2 NASA'S SPACE EXPLORATION INITIATIVE (CURRENT) 130
6.2.1 OPN as a mission-mode generator 131
6.2.2 OPN integrated with other software tools 132
6.2.3 Observation on OPN's usability 133
6.3 ENHANCED GROUND TESTING POD 133
6.3.1 What is an EGT-Pod 134
6.3.2 Alternative architectures of EGT-Pod 135
6.3.3 Infer global consequences from local knowledge 136

7 DISCUSSION 141

7.1 KEY CONTRIBUTIONS 141
7.2 OPN ADDRESSING THE NEEDS IN SYSTEM ARCHITECTING 142
7.2.1 OPN implementation meets the requirements 143
7.2.2 Meta-language and qualitative methods 144
7.2.3 Meta-language and quantitative methods 146
7.3 OPN AS AN EXECUTABLE META-LANGUAGE 147
7.3.1 OPN and pattern languages 147
7.3.2 OPN as a system description language 148
7.3.3 OPN and generative modeling techniques 150
7.3.4 OPN as a simulation language 152
7.4 FUTURE DEVELOPMENT 154
7.4.1 Theory development 154
7.4.2 Tool development 157
7.4.3 Application Development 158

8 CONCLUSION 159

9 BIBLIOGRAPHY 164

LIST OF FIGURES

FIGURE 1-1 ARCHITECTING AS COMMUNICATION AND COMPUTATION 11
FIGURE 2-1 A CATEGORY WITH THREE ARROWS AND THREE OBJECTS 18
FIGURE 2-2 3T FRAMEWORK: MANAGING KNOWLEDGE ACROSS BOUNDARIES (COURTESY OF CARLILE) 21
FIGURE 2-3 AN EXAMPLE OF E-R DIAGRAM 29
FIGURE 2-4 UML'S GRAPHICAL NOTATIONS OF FOUR VIEWS 30
FIGURE 2-5 OPM'S MODELING ENVIRONMENT, OPCAT (COURTESY OF DORI ET AL.) 33
FIGURE 2-6 A RANGE OF GRAPHICAL FORMALISMS 34
FIGURE 2-7 A CELLULAR AUTOMATON RULE AND ITS GENERATED PATTERN (COURTESY OF MATHWORLD) 36
FIGURE 2-8 META-LANGUAGE AND OBJECT LANGUAGE 51
FIGURE 2-9 THE TWO DIMENSIONS OF LANGUAGE DESIGN 53
FIGURE 3-1 MAPPING FUNCTION TO FORM 57
FIGURE 4-1 THE SPACE OF MODELING LANGUAGES 66
FIGURE 4-2 AN ANNOTATED OPN 71
FIGURE 4-3 BRANCHING AND LOOPING IN OPN 81
FIGURE 4-4 GRAPHICAL MODEL OF COMMUNICATION & COMPUTATION 87
FIGURE 4-5 TOKEN CREATION ACTIVITIES 88
FIGURE 4-6 A SIMPLE RECURSION 90
FIGURE 5-1 THE MODEL VIEW CONTROLLER OF OPN 94
FIGURE 5-2 A SCREEN SHOT OF OPN SIMULATION ENVIRONMENT 96
FIGURE 5-3 USER INTERFACE ELEMENTS FOR SPECIFYING INFERENCE RULES/ALGORITHMS 101
FIGURE 5-4 VISUALIZING SYSTEM STATES IN TERMS OF PROBABILISTIC MEASURES 103
FIGURE 5-5 SOFTWARE COMPONENTS 106
FIGURE 6-1 APOLLO FUNDING BREAKDOWN 112
FIGURE 6-2 A CONTINUOUS SPACE QUANTIZED IN DISCRETE VOCABULARY 115
FIGURE 6-3 SPECIALIZED VOCABULARY FOR THE APOLLO PROGRAM 116
FIGURE 6-4 A SCREEN SHOT OF THE OPN SIMULATION ENVIRONMENT 123
FIGURE 6-5 VARYING LEVELS OF MISSION RISK 128
FIGURE 6-6 VISUALIZING A TWO-DIMENSIONAL METRIC SPACE 130
FIGURE 6-7 WORKFLOW OF THE CURRENT NASA ARCHITECTURE STUDY (COURTESY OF SIMMONS) 131
FIGURE 6-8 AN ARCHITECTURE VISUALIZATION TOOL DRIVEN BY OPN’S OUTPUT (COURTESY OF SIMMONS) 132
FIGURE 6-9 THE EGT-POD AND ITS CARRYING VEHICLE (COURTESY OF CHRIS ANDERSON) 134
FIGURE 6-10 TWO COMPETING ARCHITECTURAL ALTERNATIVES 135
FIGURE 6-11 PROBABILISTICALLY INFER GLOBAL EFFECTS GIVEN LOCAL KNOWLEDGE 137
FIGURE 7-1 OPM'S DIFFERENT LINK TYPES 149

1 Introduction

This thesis concerns a domain-neutral meta-language that automates certain mechanical tasks in architectural reasoning for complex socio-technical systems. In this thesis, the term "architecture" denotes the stable properties of the system of interest [1-3]. The phrase "architectural reasoning" is therefore defined as a transformative process that utilizes knowledge about stable properties in a system to achieve certain global objectives. The phrase "complex socio-technical systems" refers to systems involving multiple stakeholders and requiring multiple knowledge domains. Practical experience and well-known research literature have demonstrated and articulated the advantages of designing complex socio-technical systems using architectural reasoning techniques [4-8].

Consequently, architectural reasoning techniques have flourished in various domain-specific disciplines. Civil structures, computer hardware and software are among many other domains that have developed disciplinary-specific architectural reasoning techniques. In the process of architecting complex socio-technical systems that involve multiple knowledge domains, translating domain-specific vocabulary and transferring information across different knowledge domains often becomes a considerable challenge [9]. This challenge presents two interrelated research opportunities. First, certain domain-independent architectural reasoning techniques, such as computationally intensive simulations and model generation techniques, can be leveraged over multiple disciplines [10]. Second, identifying a common language across multiple disciplinary domains can help architects and other stakeholders communicate with people and organizations outside of their domain expertise. Therefore, the objective of this thesis is to formulate a domain-independent reasoning technique in terms of communication and

computation. The aim is to design a general-purpose meta-language that specifies and automates the mechanical elements of communication and computation tasks in architectural reasoning.

This chapter presents:

- Background that motivated the development of a meta-language for system architecting
- Aims of this research work
- Research approach
- Structure by which this thesis is documented

1.1 Background

In 1911, Alfred North Whitehead wrote:

> *"Civilization advances by extending the number of important operations which we can perform without thinking about them."* [11]

Since 1911, our capacity to automatically perform a large number of important operations has increased dramatically. The tasks of conceiving and designing large-scale socio-technical systems have become increasingly difficult. As rapid advancements in component-level technologies change the way we implement and operate large-scale systems, the corresponding system design and management problems are becoming intractable due to an increasingly large number of interacting factors [10]. To extend the number of thorough and effective reasoning operations without thinking about them, we need to streamline the tasks of ***communication*** and ***computation***, so that thinking can be distributed and verified through multiple organizational levels and physical scales. The complexity of reasoning through an architectural decision is not only a quantitative problem, but also a qualitative problem. These interacting factors may be

related to different knowledge domains and could be interpreted differently under different contexts. Making architectural decisions for a socio-technical system is analogous to comparing apples and oranges: it is hard to arrive at a stable answer under the influence of multiple stakeholders. The quest to arrive at critical architectural decisions are challenged by: [12, 1, 13-16]

1. *Limited resources to fully analyze the impact of intentional decisions*
2. *Knowledge and experience about the actual system is scarce*
3. *The operating environment is entrenched with high degree of uncertainty*

At the same time, the guiding forces in the reasoning processes are: [17, 18, 15, 19-23]

1. *Stakeholder defined metrics*
2. *Anticipated performance range for the system*
3. *Available implicit and explicit knowledge*

The process of system architecting can hardly be executed as an isolated sequence of analytical routines. Instead, the decision process can be more suitably described as an evolving set of interactive events [24]. These interactive events may take place concurrently and trigger multiple iterations of deliberation cycles before reaching an architectural decision. Figure 1-1 provides a visual representation of this recursive process.

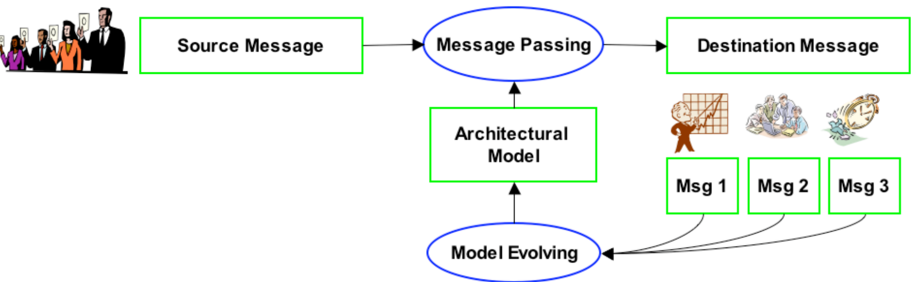

Figure 1-1 Architecting as communication and computation

This iterative process in system architectural decisions can be illustrated as a recursive composition of communication and computation activities. Figure 1.1 shows an interactive model of architectural decision-makers and other stakeholders. In this diagram, each instance of architectural decision is modeled as a "source message", it can be transported to a group of stakeholders through a "message passing" process. The stakeholders then interpret the architectural decision as a "destination message". The "destination message" may differ from the "source message" because the "message passing" process might inject noise or the message might be interpreted differently due to a shift in temporal and spatial contexts. The "destination message" may stimulate stakeholder reactions and invokes new messages to be propagated through the system. It may incur changes to the architectural model, or propose new architectural decisions. New instances of architectural decisions are then propagated through the system as new instances of "source messages" and henceforth create iterative cycles of communication.

Messages inducing changes to the "architectural model" can be thought of as instructions in a computation process. The changed "architectural model" may change the underlying model of communication that controls the "message passing" process, and therefore produces different "destination messages". This may cause a cascade of iterative computation and communication activities throughout the whole system.

In the model of communication, the "messages passing" process and the evolving "architectural model" can be viewed as a channel of communication among stakeholders. In the model of computation, the "source message" and the "model evolving" process can be viewed as an execution mechanism for an architectural model construction program. To reason through the interactive consequences of communication and computation, architects need to simultaneously improve performance on two fronts:

1. *Effectively communicate and negotiate system level consequences with a large number of interacting stakeholders.*
2. *Effectively compute and/or assess the emerging consequences of subsystem interactions across multiple abstraction layers and physical scales.*

To tackle the first challenge, a reliable and precise model of communication is needed to establish a set of efficient protocols for knowledge exchange between stakeholders. Simultaneously, architects also need a flexible and practical model of computation to compile knowledge derived from stakeholder interactions and other sources to support architectural decisions. In designing large-scale socio-technical systems such as NASA's interplanetary transportation systems, architects need an adaptive and practical instrument to improve the quality of communication and computation. Due to various theoretical and technical barriers, which will be further discussed in Chapter 2, a general-purpose instrument for architectural reasoning has not yet emerged.

1.2 Aim and Objectives

The aim of this thesis is to provide a solution for the automation of the architectural reasoning tasks in large-scale socio-technical systems. The objectives of this thesis are to

1. *Identify the needs in architectural reasoning*
2. *Propose a meta-language to address those needs*
3. *Implement the meta-language as an automation tool*
4. *Demonstrate the use of the meta-language*

1.3 Research Approach

Our research approach is presented here in three phases: theory development, tool implementation and application illustration.

1.3.1 Theory

Existing methods and tools in constructing system models were surveyed. In various application contexts, each of these tools and methods partially support the three tasks in system architecting: represent the space of architectural alternatives, generate instances of architectural alternatives, and establish a preference order among known instances of architectural alternatives. To tackle the three tasks in an integrated manner, we found that an executable meta-language can adaptively support all three tasks in various application contexts. In order to demonstrate that this meta-language-based method can be generally applied to a wide range of system architecting tasks, we need to prove three things. First, the meta-language itself must be able to describe the structures and behaviors of arbitrary systems in finite terms. Second, we need to show that all

finite-sized models can be fully enumerated within finite time. Third, we need to adaptively control the enumeration mechanism to derive analytical results within affordable computational resources. We proposed a simple meta-language schema, with three basic linguistic primitives, and demonstrated that it satisfies all three criteria. From a computational perspective [25], this meta-language approach is not restricted to specific application contexts. Therefore, it provides a generally applicable computational reasoning framework to support multiple applications domains.

1.3.2 Tool

A practical implementation of our meta-language is necessary to verify and validate its conceptual benefits. Based on our proposed language schema developed in the theoretical phase, we implemented an executable meta-language, Object-Process Network (OPN) as an instrument for architectural reasoning. OPN is designed as a communication medium between machines and humans. Its small but highly extensible vocabulary provides a standard protocol to share structural and behavioral specification across different machines. A graphical user interface is designed to reduce the amount of mental effort required for domain experts to exchange their domain-specific expertise. We partially adopted the graphical formalism of Object-Process Methodology (OPM) [26], and developed a visual modeling and simulation environment to represent, generate and evaluate architectural alternatives.

1.3.3 Application

Three examples are illustrated in OPN to demonstrate the benefits of a meta-language based architectural reasoning approach. We used the Apollo Program as a retrospective example to explore our assumptions about architectural reasoning. We applied OPN to specify the space of mission architectures, enumerate alternative mission modes, and perform tedious model construction and metric calculation tasks. The OPN tool is also applied to an ongoing project sponsored by NASA to analyze mission mode alternatives for Earth-Moon-Mars space exploration. To demonstrate that OPN can also reason about static configurations of a system, an aircraft-based testing system configuration called Enhanced Ground Testing Pod (EGT-Pod) program is also illustrated in this thesis.

1.4 Thesis synopsis

Chapter 1 is this introduction.

Chapter 2 reviews a series of interrelated theories and techniques that shaped the current paradigm [23, 27] in the field of system design and architecting. We will particularly point out that the needs in architectural reasoning are strongly associated with the needs to establish a flexible and efficient reasoning instrument, such as a meta-language.

Chapter 3 describes the needs, requirements and solution profile of an executable meta-language designed to address the current needs of systems architecting.

Chapter 4 presents the formal language schema and the language execution mechanisms of the meta-language, Object-Process Network (OPN).

Chapter 5 describes the software engineering aspects of OPN. We will explain the design rationale, the user interface design strategy, and how a simple language schema influence the software implementation activities.

Chapter 6 presents a retrospective application of our meta-language approach to NASA's Apollo Program. We used OPN to construct reasoning models, and perform metric calculation based on similar assumptions used in the original program. We also discuss how this method is being applied to the latest space exploration programs. This chapter also includes the third example about static aircraft configuration.

Chapter 7 presents the contribution of this thesis. It compares our solution with existing methods and instruments designed for architectural reasoning. We specifically compare the features of OPN and what are the unique opportunities this instrument could offer to streamline a wide range of reasoning tasks in the architectural decision-making process.

Chapter 8 summarizes the thesis's conclusions.

The following section will discuss the prior art that is directly relevant to our research.

2 Literature Review

This chapter describes the prior art related to the representation and analysis of complex socio-technical systems. This chapter is separated into two sections:

- *Theories and methods of system design and architecting, and*
- *The languages that are used as instruments to improve the reasoning activities in system architecting.*

2.1.1 Theoretical Foundation

In "Anatomy of Large Scale Systems", Moses pointed out:

> *"The mathematical field of abstract algebra can provide a language for discussing systems taken as a whole. A structured or anatomical view of engineering systems when coupled with concepts and intuition from abstract algebra can give us a relatively deep understanding of certain system issues, such as flexibility."*
>
> *Joel Moses, 2002 [28]*

The ability to design a flexible system is limited by the flexibility of the underlying modeling language. According to Moses, abstract algebra as a modeling language permits architects to flexibly and economically change the model of a complex engineering system to match the evolving analytical needs. Without a holistic modeling language, the cost of model construction and the effort required to integrate various system models may present critical concerns to be reflected in the resulting system architecture.

The power of abstract algebra comes from its ability to reason through the logical consequences of system interactions using a concise and consistent set of mathematical axioms and notations.

This formal language can also be visualized in a diagrammatic form, called Category Theory. Category theory [29, 3, 30, 31] can be thought of as the graphical representation of abstract algebra. It provides a diagrammatic language to reason about the relationships between classes of mathematical objects, not just the individual instances of numeric or symbolic values. Each category is a diagram made of a set of "objects" and "arrows". Figure 2-1 is a category with three "arrows" that represents three functions from their respective source and target "objects".

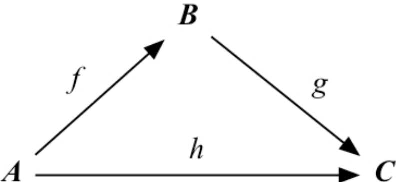

Figure 2-1 A category with three arrows and three objects

The objects ***A***, ***B***, and ***C*** represent domains of possible values. They can be in either qualitative or quantitative value domains [32]. Arrows *f*, *g* and *h* in this category represent "total functions". (A total function is a function that defines output values for all its input values. We will refer to them as functions unless otherwise noted.) Category theory utilizes the structural information of a graph to illustrate how functions can be composed or decomposed into other functions. The category shown in Figure 2-1 indicates that the function *h* can be decomposed into two functions *f* and *g*. Conversely, *f* and *g* can be replaced by one function *h*. System modelers can apply these compositional rules to create and refine functions, expose and hide the internal structures of a category. Furthermore, categories with complex internal structures can be modeled as "objects" and further composed into a higher-level diagram for mathematical analysis. It provides a recursive mechanism to organize and compress a system of functions and abstract objects into more compact diagrams that fits the cognitive capacity of well-trained mathematicians.

The connections between abstract algebra and formal language design were formally established in the late 1960s by Dana Scott's work on Domain Theory [33]. Scott's work is useful to architectural reasoning because it provides a computational framework to reason about the "partial" or "incomplete" information about a system. However, it is challenging to introduce domain theory, category theory or abstract algebra to non-mathematicians. It requires significant amounts of training and practice to adequately utilize these mathematical languages. To make abstract algebra and category theory useful to architects of socio-technical systems, we need to preserve their formal structures, while presenting the core concepts in a more accessible format.

Ashby's "Introduction to Cybernetics"[17] provides a more accessible mathematical language that describes large-scale socio-technical systems. The book presents a mathematically rigorous "theory of machines". He used a set of intuitively understandable concepts to present the theory in terms of "what does a machine do", rather than "what a machine is". This book is useful and important because it shows that a wide range of "machines" can be uniformly modeled in a consistent mathematical abstraction. Ashby stated the two scientific virtues of Cybernetics as:

1. *"…it offers a single vocabulary and a single set of concepts suitable for representing the most diverse types of system(sic). … Cybernetics offers one set of concepts that, by having exact correspondences with each branch of science, can thereby bring them into exact relation with one other(sic)."*
2. *Cybernetics is likely to reveal a great number of interesting and suggestive parallelisms between machine and brain and society. And it can provide the common language by which discoveries in one branch can readily be made use of in the others.*

Ross Ashby [17]

The language of Cybernetics is based on concepts and structures similar to abstract algebra. It is a formal language that can be executed using a modern computer. It raises the question of whether mechanical execution of formal language statements can augment human reasoning capabilities. In the book, "Cybernetics", Wiener [34], proposed the creation of a chess machine: *"... which shall offer interesting opposition to a player at some(sic) one of the many levels at which human chess players find themselves."* In May 1997, IBM's Deep Blue, a special purpose super computer won the chess game against the reigning world champion in chess. Clearly, a chess game is a discrete-state system with bounded variability. In contrast, architects of complex socio-technical systems often need to reason about systems with both continuous and discrete variables without well-specified knowledge boundaries. Can one construct a machine that can augment architects' ability to reason through an unbounded range of variability?

2.1.2 Qualitative Methods

When facing an unbounded problem, system architects often need to integrate and compose knowledge across different domain boundaries. Carlile [9] applied concepts derived from Cybernetics to model stakeholder interaction in terms of boundary objects. Domain boundaries are categorized into three types: syntactic boundary, semantic boundary, and pragmatic boundary. The syntactic objects provide a common set of representational symbols to "transfer" information from one domain to the other. The semantic objects provide the interpretive functions to "translate" the meaning of transferred information according to corresponding domain-specific contexts. The pragmatic objects are the exchange currency for the interacting

stakeholders to "transform" the meaning of symbolic information into commensurable [27] interests that lead to certain tradeoff decisions. Carlile calls this model the 3T framework. It is visualized as a triangle in Figure 2-2.

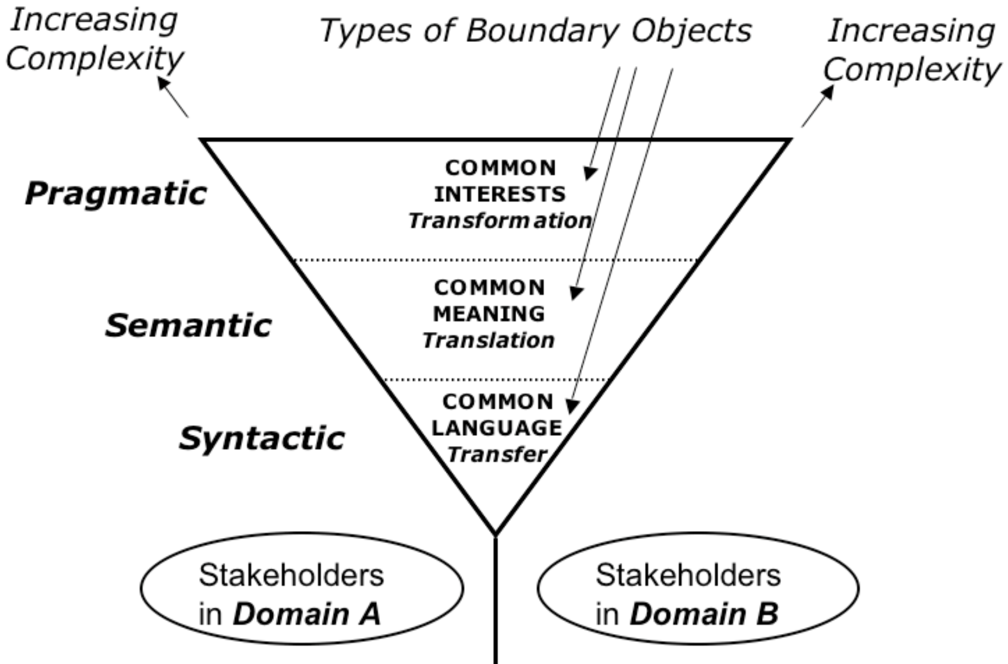

Figure 2-2 3T Framework: managing knowledge across boundaries (Courtesy of Carlile)

Carlile further characterized knowledge management as an iterative language development process.

1. *Develop a common lexicon to transfer information across domain boundaries.*
2. *Apply domain-specific knowledge to translate the meaning of the transferred information in context.*
3. *Arrive at system-level decisions by iteratively transferring and translating information between different stakeholders to negotiate tradeoffs.*

4. *Evolve the boundary objects by transforming the common lexicon and enriching domain-specific knowledge through iterative stakeholder interaction.*

According to the 3T framework, boundary objects such as email messages, computer simulation models, and physical prototypes can be treated as three instances of common languages that transfer, translate, and transform knowledge between different knowledge domains. Although the 3T framework presents a theoretical metaphor to view all artifacts in a socio-technical context as different types of languages, it does not present the concept of a domain-neutral language that can serve the purpose of communication and negotiation amongst all participating domain experts. In 3T framework terms, such a domain-neutral language would be an ideal boundary object for architectural reasoning.

2.1.2.1 Domain-neutral languages

Within the product development and process engineering communities, the utilities of domain-neutral representational techniques for describing system interactions have been broadly recognized. Design Structure Matrix (DSM) [13], Quality Function Deployment (QFD) [35], TRIZ [36], and Language Processing Method [37, 38] have been adopted by various companies and application domains. Each of these methods provides a domain-neutral language to interact with architects and other stakeholders to reason about the architectural decisions. In comparison to abstract algebra and category theory, these methods are much more accessible to business and engineering practitioners. However, these analytical methods are analytical tools only useful for the early phases in system design. They do not include a consistent and extensible mechanism like abstract algebra that can incrementally reason through layers of technical details. To conduct analysis at each different level of technical detail, a different engineering method or tools must

be incorporated to support the analytical activities. How to incorporate these application-specific methods or tools is often not specified in these methods. The architectural decisions arrived at through these methods are qualitative in nature. They are not designed to produce precise quantitative performance metrics of the system of interest. To establish a holistic architectural reasoning process, qualitative methods should be integrated with quantitative methods and quantitative performance assessment tools. It is highly desirable to create a system analysis tool that can handle both qualitative and quantitative analytical tasks.

2.1.3 Quantitative Methods

Identifying generally applicable metrics that quantifies the quality of a complex system is intellectually and technically challenging. There are many quantitative methods to reason about the design axioms or decision strategies of a system. This section describes two quantitative approaches to reason about design decisions. Suh's Axiomatic Design is illustrated here as an example of constraint-based quantitative design method. Nash's game theory is illustrated here as a quantitative method to analyze the interactive consequences of a system.

2.1.3.1 Reasoning about system constraints

Suh's Axiomatic Design [39, 40] proposes a formal framework to reason about system design. It introduces two design axioms. The first axiom is the independence axiom. This axiom leads designers to formulate design requirements in logical expressions, so that the likelihood of success of each functional requirement can be assessed statistically or analytically. It explicitly specifies the need to state independent functional requirements, so that each functional requirement specifies a design space that can be considered to be statistically independent. This

is a critical assumption in formulating a design. It serves two purposes. First, it allows designers to cover the maximum amount of design space with non-interacting design constraints. Secondly, it allows the total system's likelihood of success to be calculated using this conditionally independent assumption.

The second axiom is the information axiom. It specifies that lower content of information results in better designs. The information content of a design is a logarithm measure calculated from the likelihood of satisfying all functional requirements for that design.

$$\textit{Information content} = I = \log_2(1/p)$$

where

p : probability of satisfying functional requirement(s)

I: information content of the design

An "ideal design", in Axiomatic Design, is one that guarantees to satisfy all functional requirements. For the likelihood of satisfying the requirements to be one hundred percent, the information content, I, would have to be zero, meaning no uncertainty is involved. The information content measure establishes a preference order between design alternatives. Information content was calculated based on the joint probability of satisfying multiple functional requirements stated in conditionally independent logical statements. However, it is hard to formulate functional requirements in conditionally independent terms, therefore, calculating the joint probability of success is rather difficult. In Suh's 1990 book, "The Principles of Design", he proposed the use of a logic programming language, Prolog, to automate various aspects of Axiomatic Design reasoning tasks [39]. Suh's proposal also indicates a need in computationally supported architectural reasoning.

2.1.3.2 Reasoning about system interactions

John Nash's mathematical formulation of "Two Person Cooperative Games" [41] provided a rigorous approach to quantify system interactions in a sequence-sensitive setting. The paper defines game players' alternative courses of actions as abstract mathematical objects called "strategies." [41] How the strategies are executed in sequence is described in explicitly defined stages. These stages can be thought of as an executable process, which Nash calls the "formal negotiation model". Based on certain mathematical properties of the payoff functions defined in the game, Nash showed that his procedural model of negotiation derives the same payoff values as an axiomatic approach. This result is significant for two reasons. First, it provides a mathematical foundation to analyze sequence-dependent interactive systems. Second, and more relevant to this thesis, is that Nash demonstrated a reasoning technique that derived new knowledge by comparing analytical results from procedural specification and axiomatic rules. A procedural specification is an imperative language; it contains vocabulary and syntax to specify the sequence of activities or instructions. An axiomatic system is a kind of declarative language; its language only contains sequence-independent logical constraints. The unifying formal language that supported both aspects of Nash's analysis is abstract algebra. It further validates Moses's and Ashby's idea about how a unified language framework facilitates flexible reasoning and the discovery of new ideas. In other words, Nash's work provides a mathematical foundation for architectural reasoning, because it demonstrates how to formally represent stable properties of interactive systems in either dynamic (procedural specification) or static (axiomatic) terms. Computer scientists have utilized this insight to develop algorithms that automatically reason about tradeoff decisions in complex socio-technical systems [42].

2.2 Languages for Architecting

This section reviews existing language-based frameworks that represent, generate, and analyze alternatives for large-scale systems.

2.2.1 Pattern languages

Christopher Alexander is a civil architect who inspired the development of "pattern languages" [43] across multiple application domains. He articulated the prominent role of languages in architecting of all systems:

> *"Every creative act relies on language. It is not only those creative acts which are part of a traditional society which rely on language: all creative acts rely on pattern languages: the fumbling inexperienced constructions of a novice are made within the language which he has. The works of idiosyncratic genius are also created within some part of language too. And the most ordinary roads and bridges are all built within a language too." [1]*

As Alexander's statement implies, architects may simultaneously employ multiple languages. He initially proposed a pattern language with 253 patterns [43]. In his latest book series, he proposed a pattern language with only fifteen patterns. The newly proposed patterns are more abstract. They are also more domain-neutral than the original patterns. This trend demonstrates that pattern languages can be simplified, yet maintain or expand their expressiveness.

The use of pattern languages to reason about architectures of systems has become particularly popular in the software community [44]. The most notable use of pattern language is Software Design Patterns by Gamma et al. [45]. It provides a set of well-documented software design templates to promote design reuse. Pattern language is useful because it aggregates fine-grained

(more concrete) design tasks into coarse-grained design solutions. It reduces the system complexity by decomposing a large combinatorial design problem into a smaller combinatorial design problem based on a coarse-grained (more abstract) vocabulary. A common design vocabulary expressed in terms of patterns also helps to communicate design ideas. However, most pattern languages, software patterns included, are mostly collections of heuristic rules. They do not include a formal model of computation. They can be used as standard vocabulary in declarative languages to specify the building blocks of a design. Pattern languages rarely provide imperative information to create an overall system design. They do not specify how to compose these building blocks into a specific design instance. To better utilize pattern languages in automated architectural reasoning, one must develop computational techniques that make use of the declarative information represented by pattern languages.

2.2.2 System Description Languages

System description languages such as Entity-Relationship model (E-R model), Unified Modeling Language (UML), and Object-Process Methodology (OPM) each provide a set of syntactic rules and semantic definitions to help system architects specify a concrete composition of building blocks that represent systems in the real-world. We choose these three languages because of their distinct language design goals. E-R model provides a graphical formalism that focuses on the static structural relationships between abstract entities. UML is a comprehensive language family that presents the same system through multiple diagrammatic views, which include static, dynamic, physical assets, and human-machine interactions. OPM is a holistic system modeling methodology that subsumes multiple graphical formalisms into one diagrammatic view and one

English-like representation to complement each other to represent the structure and behavioral aspects of real-world systems. They are briefly described below.

2.2.2.1 Entity-Relationship Modeling

E-R diagram presents an intuitive and mathematically rigorous view of data. It is a widely adopted modeling technique for relational database systems. In Chen's original words:

1. *E-R model adopts the more natural view that the real world consists of entities and relationships.*
2. *It incorporates some of the important semantic information about the real world.*
3. *The model can achieve a high degree of data independence and is based on set theory and relation theory.*

One of the most extensive uses of E-R is to represent data structures. An E-R diagram is a rather mature graphical data description language. The most popular form of E-R uses three binary relationship types, they are: one-to-one relationship, one-to-many relationship, and many-to-many relationship. In Figure 2-3, each rectangle represents an entity. The numbers represent the cardinality constraints of the relationship types. One-to-one relationship binds all data entries in one entity with all data entries in the other with one-to-one correspondence. Imagine a database that stores bill of material for a car manufacturer. As shown in Figure 2-3, a car and its steering wheel have a one-to-one cardinality constraint, represented by a simple line connecting the two elements. The car and its four tires can be modeled as a one-to-many relationship, represented by a line with a "chicken feet" symbol attached to the "many" end of the relationship. Many-to-many relationship is represented through an intermediate entry. In this case, the tires of a car and the seats in the same car can be mapped through the car as a many-to-many relationship. An E-R

diagram provides a visually intuitive notation to fully specify the static data structures of a wide range of systems.

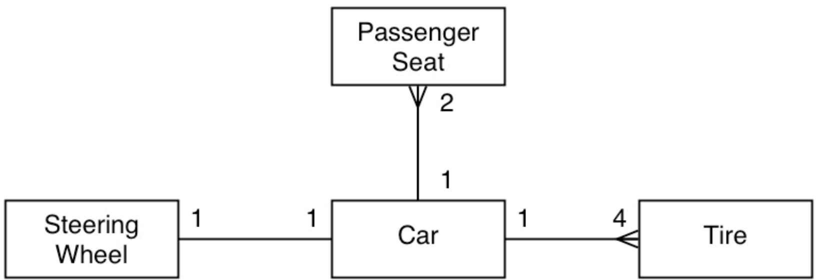

Figure 2-3 An example of E-R diagram

The E-R diagram as a declarative language can be extended to represent other types of structural relationships. It can be extended to represent specialization-generalization relationships such as class inheritance diagrams for Object-Oriented Design [26]. However, the E-R diagram only describes the static structure of a system. Most E-R diagrams ignore the dynamic aspect of a system. However, architects must be able to reason about both the static and dynamic consequences of system-level decisions. To enhance E-R models with additional knowledge about dynamic behavior of the system, additional graphical notation must be added.

2.2.2.2 Unified Modeling Language (UML)

Since 1997, UML has become a converging standard that absorbs other system description languages. Data modeling languages such as E-R diagrams are incorporated into the structural view. However, the focus of UML is still limited to software development needs. According to UML Specification 1.5, published by the Object Management Group (OMG), the non-profit organization that administers the UML standard [46]:

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.

UML was primarily designed for professional software engineers. Due to its popularity, the language design scope has expanded beyond pure software systems [47]. Its main functional views are specifically targeted at managing the concept, structure, behavior, and deployment of engineering systems. The four main functional views are: use case view, structure view, behavior view and implementation view. Figure 2-4 shows a partial set of UML graphical notations of these views.

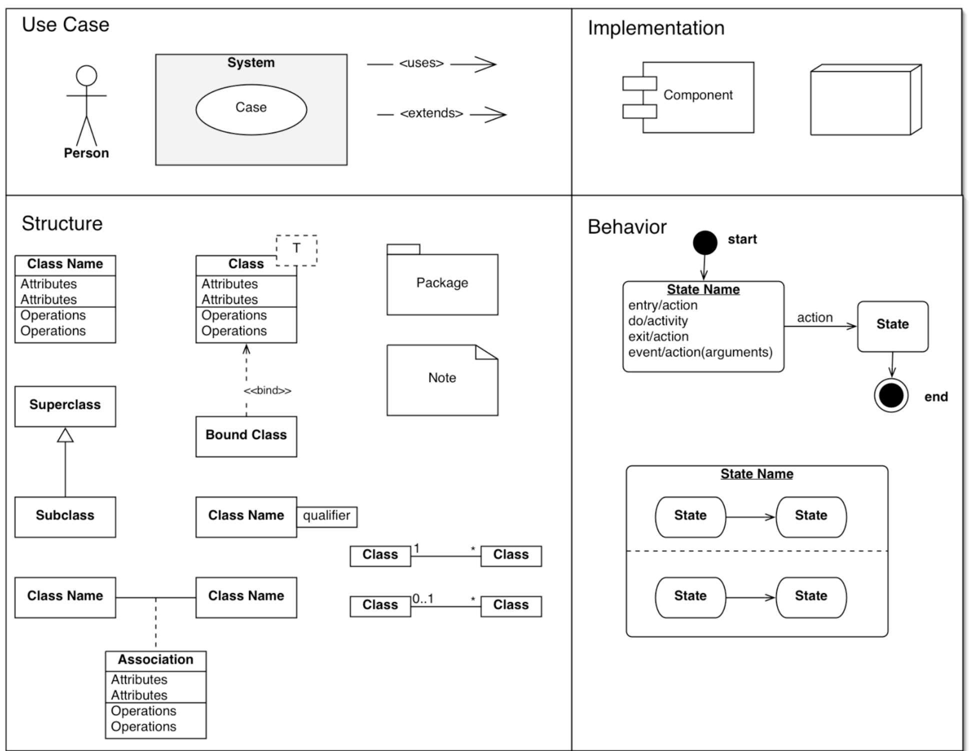

Figure 2-4 UML's graphical notations of four views

Each of these views often employs more than one diagrammatic language to visualize the relationships between different components in the view. For example, in the behavioral view, there are four diagrammatic languages to illustrate the dynamic properties of a system. They are

state-chart diagram, activity diagram, sequence diagram, and collaboration diagram. These diagrammatic languages do not have direct one-to-one semantic mapping between each other. These diagrams are designed as illustrations of design concepts; they are not inherently computable graph structures.

According to UML Specification 1.5, UML is not a programming language, it doesn't specify a run-time model, and it doesn't define an organizational process to produce software. Although different software vendors have created tools to generate executable code based on UML diagrams, they are proprietary technologies and they are not part of UML standard specification [48, 46, 47, 49, 50]

UML was originally designed for software-intensive systems. Its graphical symbols were intended to represent certain software development artifacts and activities. To represent the structures and behavior of generic systems, its language specification has undergone significant changes. In the introduction statement of UML Infrastructure 2.0 Specification (Adopted Draft copy), UML is defined as [47]:

> *"The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms."*

Expanding the scope of UML beyond the software industry may introduce additional layers of complexity to an already complex language. In UML specification 1.4, there were more than 200 different graphical primitives and 9 diagram types [48]. To tackle the language complexity

problem, the UML 2.0 tries to modularize the unified language into multiple independent sub-languages [50]. OMG also promotes concepts such as Model Driven Architecture (MDA) to manage model complexity issues by introducing a meta-model language architecture based on four layers of meta-language schemas. In other words, UML has become so complex that it needs four kinds of meta-languages to manage model abstractions. Each complexity management tool appears to introduce an additional layer of complexity. As UML grows in its popularity, the "language bloat" problem must be addressed. Otherwise, the adoption of UML would become a liability in architectural reasoning tasks.

2.2.2.3 Object-Process Methodology (OPM)

Many software engineers, researchers and the committee members on UML's revision task force have acknowledged that UML's complexity is a hindrance to system modeling [50]. Due to the size of its user community, UML needs time to incrementally refine its original language specification. In "Why Significant Change in UML is Unlikely" [51] , Dori's assessment on UML is summarized here:

1. *Model multiplicity resulting from excess diagram types and symbols*
2. *Confused behavior modeling*
3. *Obscuring influence of programming languages*

To avoid UML's Model-Multiplicity problem [52], Dori suggested that Object-Process Methodology (OPM), a visual modeling language with a single diagrammatic view and a small set of symbols, offers a superior alternative to UML. Soderborg et al. [53], demonstrated that Object-Process Methodology (OPM) can be used to specify both the structural and behavioral aspects of a system, using one language framework with two representational forms, graphical

and textual. The graphical representation, Object-Process Diagram (OPD), is a diagrammatic language as shown in the upper panel in Figure 2-5. The textual representation, Object-Process Language (OPL), is a set of formal English statements that translates the meaning of the diagram into sentences understandable by humans, shown in the lower panel in Figure 2-5. Dori and his students built OPCAT (Object-Process CASE Tool) [54], a modeling software environment, to demonstrate the feasibility and capability of this bi-modal modeling approach. Figure 2-5 is a screen shot of OPCAT.

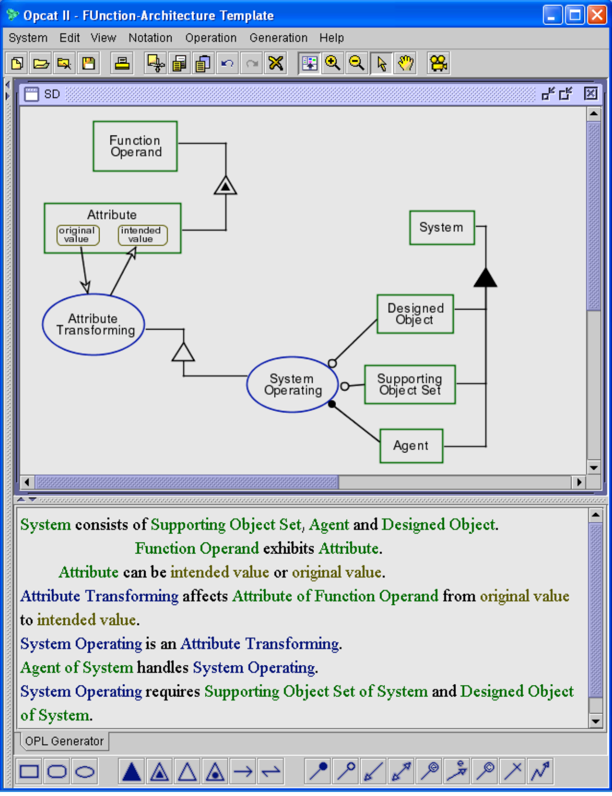

Figure 2-5 OPM's Modeling Environment, OPCAT (courtesy of Dori et al.)

OPM as a visual modeling language provides a limited set of rules to specify the precedence of process execution order. However, it does not specify a formal computational model for either discrete or continuous event systems. Nevertheless, its flexible definition of Object and Process

can be mapped onto operands and operators of a wide range of formal computational models. Koo and Fry designed and implemented a hybrid Petri-Net and Bayesian Network inference engine using OPM's graphical semantics and the Water Programming Language [55]. In Figure 2-6, we compare OPM's graphical notation against the other graphical models of computation.

Concepts \ Languages	System Dynamics	Colored Petri-Net	Probabilistic Network	Category Theory	Object-Process Methodology
Operand	Stock	Place + Colored tokens	State	Category/Functor	Object v1 v2
Operator	flow	Transition	p	Morphism	Process
Relationship	Embedded in arrows	Pre-condition P1 T1 P2 Post-condition	Conditional Probability Functions	Functors or Morphisms	
Runtime Engine	Numerical Integration Engine	Event Scheduling Engine	Belief Propagation Algorithms	Natural Transformation	S-Expression Interpreter
Application Domains	Physical System Modeling/Decision Support	Discrete Event Modeling	Reasoning about State-Space Reachability	Mathematical Reasoning, Compiler design	System Description/ Simulation
Temporal Scale	Infinitesimal Time Steps	Quantized Time Steps	A-Temporal	Abstraction of temporal scales	Multiple temporal scales
Semantic Metaphors	Dynamics of Analog Signals	Dynamics of Discrete Messages	Causal Structures	Types and Relationships	Symbolic Knowledge

Figure 2-6 A range of Graphical formalisms

2.2.3 Generative Modeling Techniques

Due to the complexity of system architecting processes, computers and computational techniques have been employed by architects to perform alternative generation tasks [56]. Civil architects and mechanical engineers have been studying shape grammar as a means to generate geometrical forms. They write computer programs to recursively apply shape specification rules to create

geometric structures [57]. The ability to use a small number of rules to create complex shapes helps architects understand and interact with the logical structures behind geometrical objects. It also extends architects' ability to reason through larger and more complex geometrical configurations in buildings and other physical objects alike. Generative shape theorists are interested in generating instances of shapes that inspire new design ideas. It is not necessary to exhaustively enumerate all possible shapes.

Shape grammar studies focus on the forms of alternative architectures. For certain socio-technical systems, the system properties may not have a direct analogy to geometrical forms. They may be better studied in the functional domain. To study the functional properties of system interactions, Wolfram [25] devised a number theoretical approach to study cellular automata by exhaustively enumerating simple automaton rules and then applying these rules to generate visual patterns. The goal is to use human perception and computer programs in combination to identify simple rules that can generate complex visual patterns. Rules that generate interesting patterns can then be further studied for their functional properties.

Wolfram's approach is applicable to architectural reasoning because it:

1. *Demonstrated that* ***exhaustive enumeration*** *of certain classes of simple rules, not the generated patterns, is computationally viable.*
2. *Studied the functional properties of simple rules by visualizing the generated patterns and categorizing them with different complexity classes.*
3. *Used a modeling language to dynamically generate, manipulate, and analyze both functions (rules) and forms (shapes) during simulation.*

An example of the rule and the generated form is illustrated in Figure 2-7.

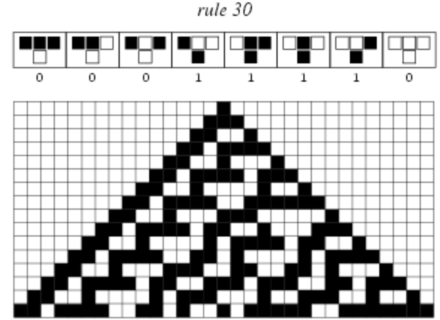

Figure 2-7 A cellular automaton rule and its generated pattern (Courtesy of MathWorld)

Figure 2-7 shows that an irregular pattern can be generated using "rule 30", the thirtieth rule out of all 256 possible rules in a rule space made of three bits [25]. Each of these rules explicitly specifies the color of the cells in the next step according to the eight possible three-neighboring-cell configurations. For example, rule 30's eight possible configurations and their color transformation results are shown graphically on the upper half of Figure 2-7. The horizontal axis in Figure 2-7 represents the spatial dimension; the vertical axis represents the temporal dimension, each row represents one time step. Some of the rules in the 256-rule space, such as rule 110, have been proven to be Turing Complete [58]. In the field of computational theory, this result is rather remarkable, because it shows that the algorithmic properties of a Turing Machine can be encoded in one simple transformation rule. This result is also an inspiration to the study of complex systems because it suggests the possibility that seemingly complex interactive phenomena may be governed by some simple and stable rules.

It is rather non-intuitive to assemble a complex engineering system using rules illustrated in pure binary forms. The instrument for architectural reasoning also needs to help architects assemble complex rules to produce simple or complex behaviors. Leslie Ann Goldberg's Ph.D. thesis studies algorithms that enumerate complex combinatorial structures [59]. Her work was to

identify algorithms that can efficiently generate combinatorial structures that are either simple or complex. The efficiency of algorithms is measured by the required amount of storage space and computing time. Her research motivation can be expressed in the following terms:

1. *Design useful algorithms for application-specific problems*
2. *Discover general methods for algorithm design*
3. *Establish an algorithmic framework to classify and generate other algorithms*

Efficient enumeration algorithms for combinatorial structures can generate both algorithms and static structures. Some of Goldberg's algorithms can be applied to reduce the required computational resources to generate architectural alternatives.

Genetic programming (GP) and genetic algorithms (GA) are analogies borrowed from the biological field to create complex systems with simple building blocks. John Holland [60, 21, 61, 22] and Karl Sims [62] have applied evolutionary rules such as crossover and mutation to dynamically generate models of complex systems. These techniques have mixed results due to the difficulties associated with constructing context sensitive fitness functions. In most cases, it is difficult to randomly sample a very large genetic space and expect good results within practical limits. To accommodate these short comings, Holland [22] proposed the concept of dynamic models, or models whose basic building blocks and executable rules may change during simulation time. For example, Kim and de Weck [63] developed genetic algorithms that use variable chromosome lengths to solve structural optimization problems. If one considers the programmatic modification of chromosome length as a way to adaptively change the size of search space, variable chromosome length GA can be considered as an algorithm that dynamically modifies its own algorithmic properties during execution. Adaptive algorithms that

evolve during execution time may need special programming language features to better support their implementation and debugging tasks. Specifically, functional programming languages are particularly designed to implement software programs that manipulate programs [64]. To study the architectural properties of complex evolutionary systems, architects often employ programming languages with dedicated features to simulate the properties of interest.

2.2.4 Simulation Languages

Zeigler [65] proposed a categorization scheme that distinguishes formal simulation models into five dimensions:

- *Continuous time vs. discrete time*
- *Discrete state vs. continuous state*
- *Deterministic model vs. non-deterministic model*
- *Autonomous model vs. non-autonomous model*
- *Time invariant vs. time varying.*

Continuous time models are models whose clock increments in infinitesimal time units. Discrete time models are models whose clock increments in integer time units. Discrete and continuous state models are models that contain discrete and continuous state variables respectively. Hybrid state models are models that contain both kinds of variables. Deterministic models contain no random variables, where non-deterministic or stochastic models contain at least one random variable. Models that are completely isolated from influences in their environments are considered to be autonomous; the opposite kind of model is non-autonomous, requiring external stimulus to perform simulation. The last categorization of model is based on whether the model

changes during its simulation time. If the rules of interaction change during simulation time, it is considered to be a time variant model. If the rule of interaction doesn't change during its simulation time, it is considered to be a time invariant model.

To model a complex socio-technical system, it is likely that all these model categories need to be combined and used at different points in the system development process. In this thesis, we will focus on three types of graphical formalisms: Probabilistic Graph Model, Petri Net, and System Dynamics. These three graphical formalisms are presented here because each of them has been extensively developed to accommodate a wide range of real-world applications. They have been extended to cover other areas of simulation needs, and therefore developed different levels of hybrid simulation capabilities within each of the three types. We will use these three basic types of graphical formalism to illustrate the state-of-the-art simulation languages.

2.2.4.1 Probabilistic Graph Models

Causal structures such as Bayesian Belief Networks [66-68], Markov Chains, and Factor Graphs [69] are graph-based models for analyzing decisions or events under uncertainty. They provide a reasoning framework for people or machines to reach rational decisions even when there is not enough information [70]. Probabilistic graph models and their supporting computational algorithms can also be used to formulate and solve N-person games based on Nash's equilibrium assumptions [41]. Jordan best articulated the power of probabilistic graphical models:

> *"Graphical models are a marriage between probability theory and graph theory. They provide a natural tool for dealing with two problems that occur throughout applied mathematics and engineering -- uncertainty and complexity. ... The graphical model framework provides a way to view all of these systems as instances of a common*

underlying formalism. This view has many advantages -- in particular, specialized techniques that have been developed in one field can be transferred between research communities and exploited more widely."

Michael Jordan, 1998

The strength in using probabilistic graphical models lies in its ease of model construction. However, solving the problem computationally has been demonstrated to be a combinatorially explosive problem [71]. Research activities in this area focus on finding efficient algorithms to solve a problem with large number of variables and states. Algorithms that find approximate solutions for large size problems have been developed [72].

Solving a Probabilistic Graphical Model is about calculating marginal probability distribution functions for each of the variables in the graph model. In 1988, Pearl first presented a "belief propagation algorithm" to solve the marginal probability calculation problem for acyclic graphs. Since then, many algorithms have been developed based on this concept, which includes the bucket elimination algorithm [73], the variable elimination algorithm [74], and the sum-product algorithm [69].

The sum-product algorithm can be thought of as a generalization of the belief propagation originally developed by Pearl [66]. It follows a simple computational rule to propagate change messages and update the marginal probability functions throughout the graph. Kschischang et al.'s [69], demonstrated that the sum-product algorithm can subsume many other algorithms such as the Viterbi algorithm, the Kalman filter, and certain fast Fourier transform algorithms. The power of this algorithm is derived from its simplicity in its

message propagation rule. More details of the sum-product algorithm can be found in Kschischang's paper.

The sum-product algorithm operates on a graphical data structure called a factor graph. Factor graph is a bi-partite graph that resembles Petri Net's bi-partite graph formalism [75], and the message passing is similar to the concept of moving tokens between places and transitions. However, there are two main differences that separate factor graphs from Petri Nets.

1. *Factor graphs only use sums and products as the two arithmetic operators that operate on probability functions. Instead, the "transitions" in Petri Nets represent operators that may represent arbitrary transformation functions*
2. *Factor graphs schedule message-passing events based on a customizable message update rule. Petri Net schedules the propagation of tokens based on the structure of the graph and the duration required to complete each transition execution event*

In the self-modifying aspect of modeling, it is particularly easy to modify Probabilistic Graphical Models because the connections between nodes in a graph are driven by statistical data, which can be mechanically retrieved through either manual or automated data feeds. Other probabilistic graph models such as Dynamic Bayesian Belief Network [76], and Learning Bayesian Networks [77] have incorporated the notion of time variant features to update both the structure and statistical data content during simulation.

2.2.4.2 Petri Nets

Petri Net is named after C. A. Petri, whose Ph.D. thesis [78], "Kommunikation mit Automaten" (Communication with automata), started a major revolution in graph-based simulation methods. Petri originally formulated Petri Net as a theoretical basis of communication between asynchronous components of computing devices. This concept was later generalized to cover the description of causal relationships between arbitrary events. Petri Net has been applied to a wide range of applications, such as workflow modeling, legal systems, distributed computing systems, manufacturing system design, and many others [79]. Petri Net is often employed as an analytical tool to study concurrent behavior in discrete event systems. Due to its extensibility, hybrid extensions such as probabilistic, fuzzy and continuous event models have been added to the Petri Net language family [80]. To encompass all the variations of Petri Net, we offer the following definition:

> *A Petri Net is a directed bi-partite graph that uses tokens to represent the state of the system being modeled. Nodes in the graph are divided into two types, passive and active. Passive nodes that store the tokens are called places. Active nodes that move the token between places are called transitions.*

The key strengths of Petri Nets can be summarized as follows:

1. *Petri Net is a rigorous mathematical representation of interacting systems*
2. *Petri Net is also an intuitive diagrammatic language that can be understood by non-mathematicians*

3. *Petri Net is a language with closure properties. Standard operators such as concatenation, union, and intersection can be applied to two or more Petri Nets and the resulting network would remain a Petri Net*

The closure property of Petri Nets is particularly beneficial in composing models for complex system. It provides the mathematical basis to automate composition of new models from existing models. Petri Nets can also be used as a generator of other Petri Nets, tokens can record the places and transitions while being moved around. The "itinerary" of the tokens can be represented as a Petri Net and stored in the tokens [81-83]. The ability to generate models through executing a Petri Net makes Petri Nets a meta-language.

Researchers in the field of modeling languages have criticized Petri Net for its biased focus toward the process or dynamic aspects of a system. In contrast, the E-R diagram, System Dynamics, and factor graphs, which can also be represented as bi-partite graphs, provide the semantic elements to express the static aspects of a system. Another weakness of Petri Net is its reliance on graphical formalism. Even a small number of places and transitions can appear visually complex. However, this problem applies to all graphical models of systems.

Petri Net has become a popular system modeling language in the real-time embedded systems community. It has been applied to other areas with much less popularity. Discounting its inherent weakness in graphical notation and its focus on system dynamics, the lack of popularity in socio-technical system modeling is related to the demographics of the Petri Net research community. Computer scientists and mathematicians wrote the majority of research papers and books on Petri Nets. These publications contain highly technical content, making them less accessible to average system modelers. When choosing a

modeling language for the analysis of socio-technical systems, an active research community that applies the language in a socio-technical context is another driving factor.

2.2.4.3 System Dynamics

Jay Forrester introduced System Dynamics as a modeling and simulation tool to the research community that studies socio-technical systems. System Dynamic models help architects of socio-technical systems to visualize the structural interdependencies among variables. The model can be executed to quantify variable interactions in a temporal context. It has been applied to many high-profile socio-technical problems, including urban planning, financial market dynamics, human population models, and product development processes [84, 85].

System dynamic models graphically depict causal relationships as arrows that connect two kinds of changing variables. One kind of variable is called "stock", they are variables that accumulate change over time. The other kind of variable is called "flow", they control the rate of change over time. Once a graphical model is constructed to represent a system of interest, a computational engine will calculate the interactive effects among variables over a specified period of simulated time. The graphical model of System Dynamics also helps analysts visualize the reinforcing and balancing loops in a network of causal relationships. The historical states of each variable are recorded and can be used to analyze the interactions between different variables. System dynamics provides an instrument to help people reason about complex and dynamic scenarios. Forrester has the following comments [84]:

> *"To deal with practical management and economical problems of pressing importance, a mathematical model must be able to include all of the categories leading to realistic representation of corporate and economic behavior. ... The model must be able to accept*

our descriptions of organizational form, policy, and the tangible and intangible factors that determine how the system evolves with time."

The mathematical model of System Dynamics is based on a system of differential equations. Users specify organizational forms, policies, tangible and intangible factors in arithmetic equations and numeric values. These values are processed through a numeric integration engine that integrates changes over time. This numeric integration approach avoids two problems. First, it bypasses the need to solve differential equations analytically, which guarantees that all models can have some numeric solutions. Second, numeric integration breaks the cycles in the graph model by taking calculated values from the previous time step. Therefore, independent of the number of cycles and the arithmetic equations that specify the dynamic values of each variable, properly initialized models can always be simulated and finished within polynomial computation time.

However, using numeric values and arithmetic equations to specify the behavior of a system is somewhat limiting. There are numerous cases where a system may need to trigger certain processes, and these events can change the structure of the model under different conditions. This is particularly difficult to express using numeric values and arithmetic equations, only. To accommodate these needs, Hines et al. [86, 87], have implemented software extensions to incorporate event-triggering mechanisms in the numeric integration engine.

The effort and skills required to construct a meaningful system dynamics model are not to be overlooked. It takes experience and time to learn about the system of interest. It takes natural talent and conversational skill to translate a qualitative inquiry into numeric variables. Then, it takes time to interpret the model and calibrate its input manually. Most System Dynamics

models are "time invariant" models. The arithmetic equations and the structures of the model do not change during simulation time. The data structures of existing System Dynamic tools do not inherently support dynamic model generation and self-modification. To quote the introduction statement of Sterman's book on "Business Dynamics" [85]: *"The greatest constant of modern time is change."* To fulfill the needs of representing changes in the real world, we need a simulation environment that can endure and specify changes during simulation time.

2.2.4.4 Textual Simulation Languages

From a human-machine interface viewpoint, graphical modeling languages help people better visualize the relationships and structures of interacting variables, but a graph with a large number of variables can be visually incomprehensible.

Back in 1965, Ole-Johan Dahl and Kristen Nygaard explicitly designed a programming language to support the analysis of socio-technical systems. In Nygaard's own words [88]:

"From the very outset SIMULA was regarded as a system description language ...

1. *The language should be built around a general mathematical structure with few basic concepts. This structure should furnish the operation(s) research workers with a standardized approach in his description so that he can easily define and describe the various components of the system in terms of these concepts.*
2. *It should be unifying, point out similarities and differences between various kinds of network systems.*
3. *It should be directing, and force the operations research worker to consider all aspects of the network.*

4. *It should be general and allow the description of very wide classes of network systems and other systems which may be analyzed by simulation, and should for this purpose contain a general algebraic and dynamic language, as for example ALGOL and FORTRAN.*
5. *It should be easy to read and to print, and should be suitable for communication between scientists studying networks.*
6. *It should be problem-oriented and not computer-oriented, even if this implies an appreciable increase in the amount of work which has to be done by the computer."*

SIMULA is also considered to be the first object-orientation language. Today's popular languages, Smalltalk, Common Lisp Object System, Java, C++, and Python [89] have all been influenced by SIMULA's original concepts in object-orientation. However, all these languages are textual languages. It is difficult to communicate the structure and the potential interactions in a complex system to non-programmers by source code of a simulation model. Ideally, textual and graphical modeling languages should be combined to leverage the strengths derived from both language types.

A unique breed of textual language is changing the way people communicate with each other. The invention of Hyper Text Markup Language (HTML) and the Hyper Text Transport Protocol (HTTP) made a significant impact to our socio-technical system [90]. The simultaneous introduction of a standard syntax along with a standard data transport mechanism profoundly changed the way we live and learn. Berners-Lee's invention demonstrated that a simple and static computer language [4], properly deployed, could unleash complex and dynamic reactions of many people, machines, and societies. Berners-Lee's more recent focus on Extensible Markup

Language (XML) and Semantic Web Initiatives [91] further pushed the concept of sharing a common syntax across multiple knowledge domains.

Many researchers and software developers have been converging toward the use of XML as a standard syntax for textual languages. Various middleware software vendors have adopted XML as a data-encoding standard to integrate their workflow systems, organize corporate information repositories, and compose collaborative simulation experiments [20, 92, 93].

However, one must point out that XML was designed to encode static data. It was not intended to encode dynamic procedures. Researchers and software companies have made serious attempts to encode procedural knowledge in XML [94, 55]. Partially due to XML's verbose syntax, an XML-based general-purpose modeling and simulation language has yet to become popular.

2.2.5 Meta-languages

Meta-languages and meta-programming are well-known solution patterns in mathematical reasoning [95] and in the computer science literature [96]. In software engineering, meta-language and meta-programming are often applied to generate code and design compilers for programming languages. Czarnecki and Eisenecker clarify the nature of meta-languages [96]:

> *"The word 'meta' is borrowed from the Greek word meaning "after" or "beyond" and is used to denote a shift in level. ... In linguistics, the term "meta" does not imply anything speculative or mysterious, it simply implies the relationship of 'being about' something, for example, a meta-language is a language to describe another language. English grammar is a meta-language with respect to some text written in English because it explains its structure. The*

usage of 'meta' in linguistics corresponds to its usage in computer science, where meta-programs are programs about some base-level programs."

Based on this definition, a meta-language is simply a language that specifies the grammatical structure and vocabulary of other languages. A language being described by a meta-language is referred to as the "object language". An object language can be used to describe another language, so that it can become a meta-language in turn for its object language. Church [95] and Carnap [97] have a similar definition of meta-language and object language. Their definition is summarized below.

Definition: *Let **M** and **O** be two languages. If **O** can be described using vocabulary and grammatical rules available in **M**, then **O** is the "**object language**" and **M** is the "**meta-language**", and vice versa.*

Milner created an ***executable*** meta-language, ML [98]. It was originally designed for mathematicians and computer scientists to perform automated theorem proving tasks. Programming language designers have been using ML to specify the syntax and semantics of other object languages and create instances of executable programming languages using ML. Other languages such as Lisp [12, 99] and Haskell [100] also support similar meta-language features. Meta-languages are often designed and implemented as general-purpose programming languages. They support declarative language features to describe arbitrary systems. They support imperative features to specify algorithms and execute calculation tasks. They are particularly suitable for generative modeling techniques, because they can manipulate data structures as well as algorithmic specification of other programs. However, executable meta-languages such as ML, Lisp and Haskell are not designed for casual users. They all require

significant programming experience to internalize the syntax and semantics of these programming languages. Although, executable meta-languages present many language qualities that are desirable in architectural reasoning, it is unlikely that system architects and stakeholders can directly utilize existing meta-languages as a common medium for both communication and computation.

2.2.5.1 Functions of an executable meta-language

The function of an executable meta-language is to automatically manipulate representational schemes, enumerate combinatorial possibilities, and perform mechanical calculation tasks. An executable meta-language often serves the functions of: communication and computation, including recursion.

1. *Communication: meta-languages are the instruments to define a common data structure that relays qualitative and quantitative information across machines, individuals, and organizations.*
2. *Computation: meta-languages are instruments that map domain-specific knowledge to computable rules, so that people and machines can follow the formal mapping to interpret and execute instructions encoded in meta-language models.*
3. *Recursion: meta-languages often recursively apply the same set of rules and symbols to adaptively define data structures and computable rules in varying context. Recursion allows a meta-language to define other languages in the most efficient way possible.*

Figure 2-8 shows the meta/object language relationships between two well-known programming languages.

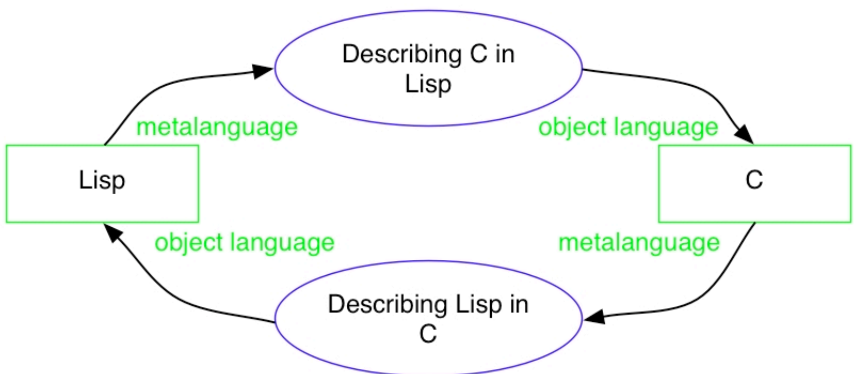

Figure 2-8 Meta-language and object language

The property of this introspective language definition structure is structurally similar to systems that are recursively composed of other systems. Therefore, meta-language as a class of modeling languages naturally serves as the formal representation of "systems of systems". Architects can conveniently model a system of systems based on a system of languages.

The practical advantage of defining a root meta-language is to provide consistency. Adopting a root meta-language provides a unified representational foundation to construct layered models for real systems, so that users of this language can incrementally tackle the complexity of system interactions, without losing sight of the possible connections between various sub-systems.

In principle, the recursive or self-referential nature of meta-languages makes them pervasive in our daily thought process; one cannot think without it [101-104]. On the down side, the multiple levels of recursions can be counter-intuitive to people and therefore intentionally avoided in practical use. To break away from this dilemma, a general-purpose meta-language must be intuitive, so that users can utilize meta-languages at multiple levels of abstraction, without confusion from the depth of recursion. Then, architects and other stakeholders can better focus their mental effort on creative activities that are not yet computable.

2.3 Summary of reviewed modeling languages

This section presents a summary of our literature review on modeling languages. We first present Jorgensen's comparative results on existing modeling languages. Then, we will present a two dimensional diagram to summarize the space of modeling languages for system representation.

2.3.1 Comparative Studies of modeling languages

Jorgensen [105] conducted an extensive study on modeling languages that included UML, System Dynamics, Petri Nets, and other textual, informal, or semi-formal process languages. We cite his original words:

1. *Many languages are complex, containing numerous types and views not integrated in a systematic manner. This is especially the case for UML.*
2. *In many cases mathematical, logical or technical concepts are applied instead of user or domain oriented (needs). Petri Nets and constraint-based languages exemplify this.*
3. *The languages that are precise and formal enough for automatic execution offer few opportunities for human contributions to interactive activation. The languages do not handle process models with varying degrees of specificity.*
4. *The semantics of language elements is generally static and not easily adopted to local context or multiple perspectives.*

Jogensen's statements echo our own observations. Our literature review indicates that system architects are still in need of a modeling language that is simple, intuitive, and executable to support many tedious tasks in architectural reasoning.

2.3.2 Emphasis of modeling languages

Existing modeling languages are not adequate for holistic system modeling because they are designed to emphasize certain aspects of modeling for their respective application domains. Figure 2-9 shows a two-dimensional categorization of language properties.

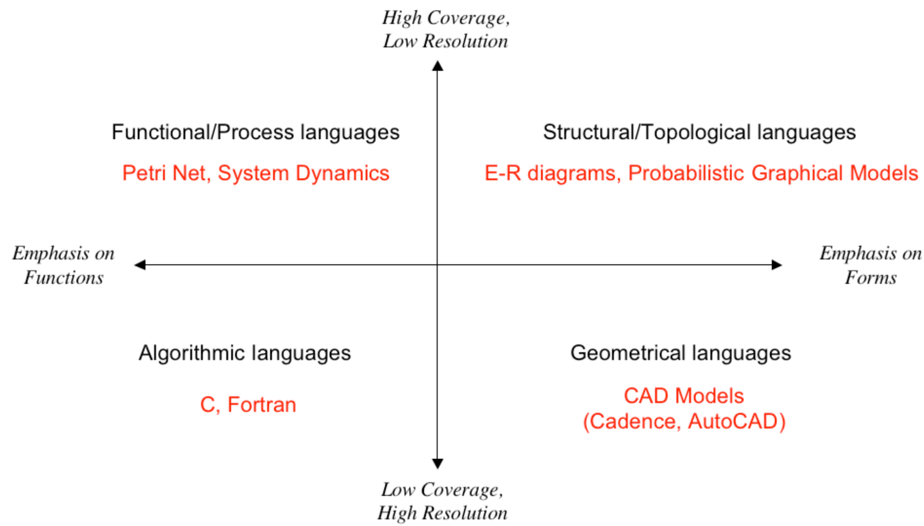

Figure 2-9 The two dimensions of language design

The horizontal dimension in Figure 2-9 represents the function vs. form emphasis. For example, Petri Nets and System Dynamics are designed to emphasis the functional or process aspect of systems. In contrast, E-R models and Probabilistic Graphical Models are mostly interested in the structural relationships between their components; their modeling emphasis is on the forms or structural configurations of the system. The vertical dimension in Figure 2-9 represents the tradeoff between coverage and resolution. For example, modeling languages such as E-R diagrams often choose words that cover the entire class of things, such as “Car” and “Steering Wheel”. In contrast, modeling tools for geometrical objects such as AutoCAD and Cadence,

must explicitly specify the shape and geometrical configuration of a car or an electronic circuit at a high level of resolution. At the same time, each high-resolution model looses the expressiveness to represent a generalized class of things. On the process side of Figure 2-9, it shows that programming languages such as C and Fortran are designed to precisely describe the algorithmic properties of dynamic systems. However, system architects might not choose them as a communication medium to illustrate the dynamic properties of socio-technical systems.

We intentionally left out UML and OPM in Figure 2-9, because they both are trying to cover the entire two-dimensional language space. UML is a modeling language that intends to cover the entire language space by adding many instances of sub-languages. This ultimately led to its complex family of languages. In contrast, OPM tries to assimilate different languages into a single diagrammatic view and a matching textual representation. It avoids the model-multiplicity problems in UML. However, it also faces the danger of introducing too many semantic elements into a single set of notation, and lead to notation bloat within one language. These language design tradeoff questions help us rethink system architects' language needs.

Executable meta-languages present many desirable features for modeling complex socio-technical systems. They are designed by mathematicians and computer scientists who are skilled in abstract algebra, category theory, and domain theory. The weaknesses of existing meta-languages are related to their abstract syntax and programming semantics. It is a serious challenge to design an executable meta-language that preserves its flexibility, while making it accessible to architects and stakeholders who are not professional programmers. This challenge is the main focus of this thesis.

The following chapter will describe the needs and requirements that lead to the design and development of an executable meta-language, Object-Process Network.

3 Needs and Requirements

This chapter describes the needs, requirements and solution profile of an executable meta-language for system architecting.

3.1 The needs of systems architecting

Section 1.1 presented the motivation of this research. These language needs of system architects can be summarized here.

1. *Communication between stakeholders and architects and among architects in different knowledge domains;*
2. *Computationally assess the consequences of system interaction at various layers of abstraction, in various knowledge domains and at different physical scales.*

Based on our literature review (Chapter 2), we found that a holistic modeling language has yet to be designed and implemented. Each of the modeling languages we studied has only been able to partially fulfill the needs of system architects of socio-technical systems. Since the underlying mechanisms by which we create models and perform simulations have not changed significantly since the late 1960's, the effort involved in changing and sharing models of a complex socio-technical system remains unchanged. To avoid this dilemma, we propose the following features of a holistic modeling language:

1. *It must be simple, yet flexible*
2. *It must be mathematically rigorous, executable, and easy to understand.*
3. *It must support self-modifying features and be extensible to a suite of domain specific needs*

4. *It needs an integrated user interface that combines both graphical and textual languages*
5. *It should adopt a standard syntax for sharing models between different computers*

3.2 Three types of architectural reasoning tasks

There are three distinctive reasoning tasks in system architecting: modeling architectural alternatives, generating instances of architectures, and calculating performance metrics for each architectural instance. Figure 3.1 is an Object-Process Network (OPN) that depicts the three architectural reasoning tasks.

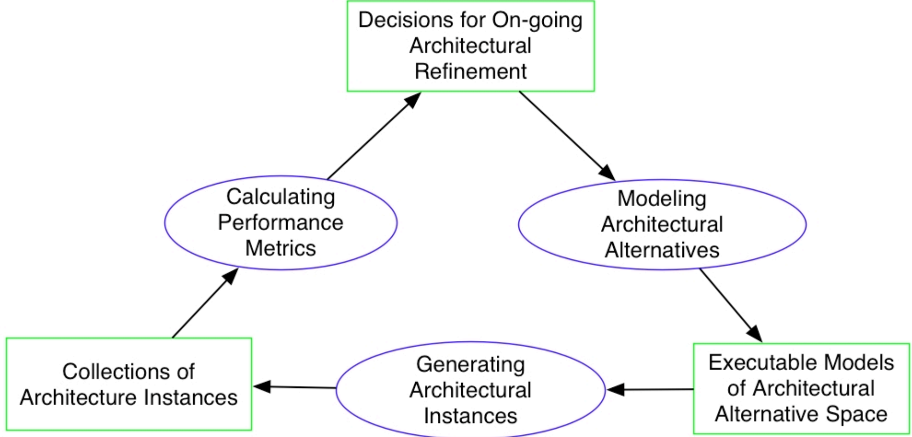

Figure 3-1 Mapping Function to Form

The top box in Figure 3-1 indicates that architects and key stakeholders must present certain decisions to direct the exploration effort in the massive combinatorial space of architectural alternatives. They need to create a model of architectural alternatives that prescribes the space of alternatives. This part of the architectural modeling effort provides the declarative knowledge to shape the space of architectural alternatives, so that architects and stakeholders can utilize this knowledge to generate instances of architectural solutions.

In Figure 3-1, the box in the right-hand corner suggests the models of architectural alternatives are executable. An executable model is a program written in an imperative modeling language that can instruct a machine to mechanically generate architectural instances. When the number of architectural instances is too large, the modeling language should allow architects to adaptively control the number of generated architectural instances based on practical needs. When the space of architectural alternatives is continuous or uncountable, this modeling language should provide some classification scheme to systematically organize the space into a set of distinctive architectural solutions, so that an imperative program can be executed to enumerate the entire space of architectural solutions in numeric or symbolic terms.

The box in the left-hand corner in Figure 3-1 represents a collection of architectural instances generated by the instance enumeration process. To select an instance of architectural solution or proceed with further investigative activities, we need a way to calculate the performance metrics for each of the architectural instances. Since each architectural instance is represented by a set of static numerical or symbolic values, they can be used as inputs to construct performance metrics calculation routines that properly reflect the structural and behavioral characteristics of each instance. The process of evaluating the performance metrics for each instance of system architecture is called system performance simulation. The performance simulation results of a collection of architectural instances provide a rational source of information to reach ongoing architectural decisions.

This section shows that architects need a language to first declare the space of alternatives, then, the language also needs to provide the imperative information to specify the process of generating instances of alternatives. To reach an architectural decision or to decide to pursue

further architectural investigation, architects need a language that can faithfully simulate the structural and behavioral properties of the proposed system of interests.

3.3 Requirements of the architects' meta-language

To support the three types of reasoning tasks illustrated in Section 3.2, an executable meta-language for system architecting must:

1. *Formally represent and specify the space of architectural alternatives by reflecting the knowledge of system variability across multiple knowledge domains*
2. *Automatically generate, enumerate and encode all instances of architectural alternatives specified in the meta-language*
3. *Adaptively calculate metrics associated with each generated architectural instance to help architects and other stakeholders perform tradeoff analysis on all instances of architectural alternatives*

The following subsections describe other required features of an executable meta-language for system architecting.

3.3.1 Subsume various models of computation

Designing and architecting a system can be characterized as a process of simulating future events [106]. To assess the interactive consequences of various socio-technical factors, the simulation semantics must be able to represent the range of structural and behavioral abstractions. As more knowledge about the system of interest accumulates, the meta-language must be able to flexibly represent the range of structural and behavioral formalisms. It is a well-known property that all

Turing-Complete languages can represent other formal languages through a complete and finite translation mechanism. This property is necessary for the meta-language to incorporate new language formalism without reaching an inherent syntax or semantic limitation. Therefore, we need to show that our meta-language is Turing Complete.

3.3.2 Generate possible subsets of alternatives within finite time

The thoroughness or completeness of alternative analysis is defined by the ability to exhaustively study all possible alternatives. Two immediate challenges arise. For systems that involve continuous state variables, the number of combinatorial states is uncountable by definition, making it theoretically infeasible to enumerate all the possibilities. For systems described with discrete qualitative and quantitative variables, the size of combinatorial possibilities can easily overwhelm any available computational resources. To establish a theoretical framework that can be generally applicable to the study of design alternatives, we need to specify the conditions in which exhaustive alternative space enumeration is possible.

3.3.3 Adaptively construct computable expressions

Each instance of alternatives generated by the executable meta-language is an instance of an object language. Therefore, each generated alternative instance must contain the syntactic and semantic information to specify the computational behavior of an executable object language. This information can be utilized to construct computable expressions that can evaluate performance metrics for each of the generated alternative instances. Without an automated model construction mechanism, the large number of enumerated alternatives could not be evaluated in

all their unique structural and functional features. We need to utilize the structural and functional properties in the "object languages" to adaptively construct metric calculation models, so that the structural and functional properties of each instance of alternatives can be fully investigated. This model construction mechanism is a critical feature to thoroughly evaluate the full range of architectural alternatives. By ignoring some structural and functional aspects of individual alternative instances, many critical architectural consequences may be overlooked. A meta-language-based simulation environment needs to preserve and utilize all the structural and functional models of individual alternative instances.

3.3.4 Enable Model Introspection

The notion of introspection [99] is a basic concept in the design of executable meta-languages. Introspection is the capability to programmatically examine and manipulate the internal parts of a program. This includes local variables, and algorithms. A simple and consistent language model would better facilitate language introspection. We achieve this by the use of one pair of meta-operand and meta-operator as the atomic units of data and computation. It provides a unifying mechanism to access data and procedures across any part of the language system. This feature is particularly useful to enable data and algorithm sharing across different object languages that are specified through the same meta-language. It provides a consistent mechanism to reduce representational redundancy, thereby reducing potential errors. Introspective data and procedural structures also help condense the size of the language kernel, making it concise and portable to various computing and communication environments.

3.3.5 Support layered abstraction models

To support the analysis of complex systems, organizing different parts of the system into different layers of abstraction and classifying system attributes into different scales are important decomposition techniques to modularize model complexity. In formal representation of systems, three layers of abstract semantics can be classified as: declarative rules, imperative procedures, and customized simulation code. Declarative rules are formal statements that do not explicitly specify the sequence of event occurrence. For example, the expressions "A>B" and "B>C" are conditional statements; they do not need to be evaluated in a specific order. Due to their sequence-independent nature, declarative rules are particularly effective at specifying system properties that do involve the notion of precedence order, such as the structural relationships between system attributes. Imperative procedures on the other hand, provide explicit information to specify the sequence of event occurrence. They are effective at specifying the dynamic behavior of a system. Simulation code is the domain-specific library that enriches the capability of a general-purpose simulation environment. The semantic model must allow both declarative and imperative statements to conveniently utilize features offered through the library of simulation code.

3.3.6 Diagrammatically represent system models

Reasoning through diagrams is a typical technique employed by people across all disciplines. Larkin and Simon [107] best summarized the rationales of diagrammatic reasoning as follows:

1. *Diagrams can group together all information that is used together, thus avoiding large amount of search for the elements needed to make a problem solving inference.*

2. *Diagrams typically use location to group information about a single element, avoiding the need to match symbolic labels*
3. *Diagrams automatically support a large number of perceptual inferences, which are extremely easy for human(sic).*

Diagrammatic representation of our meta-language is desirable because it can provide a cognitively appealing, consistent, interface for technical and non-technical users alike. The syntax of the meta-language can be visually displayed and the execution procedures of the meta-language can be animated. These computer visualization techniques may reduce the mental labor required to construct and debug a complex model.

3.3.7 Deploy across standard computing platforms

In a world of computation and communication, meta-languages and their relevant technologies are pervasive. Java Virtual Machine (JVM) [108], Extensible Markup Language (XML) [94], Resource Definition Framework (RDF)[109] , Common Language Runtime (CRL) [96] environment, and other machine processable meta-languages all have an identical goal: provide an extensible language kernel to support a wide range of applications in various physical environments. The software implementation aspect of the meta-language is important because it affects the ability to deploy the computational and communication services to coordinate architects and other stakeholders across multiple geographical locations and time. A compact language kernel with a small number of linguistic primitives is ideal, because it reduces the minimum resource requirement, therefore making it feasible to deploy onto the widest possible collection of machines. Utilizing popular software standards such as XML and Java is also

important because it helps us leverage a rich set of software features that are embedded within standard platforms or accessible through the open-source community.

3.4 A Solution Profile

In order to meet the requirements listed above, we propose our meta-language to include the following features:

1. *Specify an executable model of communication and computation*
2. *Satisfy Turing Completeness*
3. *Support a three-tiered semantic model to reduce language complexity*
4. *Use diagrams to visualize model structure and behavior*
5. *Use one meta-operand and one meta-operator to build the language kernel*
6. *Utilize technologies supported by standard platforms*

This chapter presented the requirements and fundamental concepts of a meta-language for systems architecting. Chapter 4 describes OPN, the meta-language we propose in this thesis, and demonstrates how it meets these requirements.

4 An executable Meta-Language: Object-Process Network

This chapter describes Object-Process Network (OPN), an executable meta-language for systems architecting. We choose the name OPN for two reasons. First, we want to acknowledge Dori's work on Object-Process Diagram (OPD) that avoids the bias toward either pure Object-Orientation or Process-Orientation in system modeling. Second, we want to distinguish OPN from OPD because OPN strictly follows a bi-partite network structure that only allows direct connections between Object-Process pairs. In contrast, OPD defines many types of connections to directly connect Object to Object and Processes to Process. The refined graphical syntax in OPN makes it easier to implement execution engines based on other graphical computational models that follow the bi-partite graph formalism.

4.1 The space of modeling languages

To perform architectural reasoning tasks, architects must visualize the interactions between objects and processes, and represent both the space of alternatives and architectural instances by choosing a language that can adaptively cover all the squares in Figure 4-1. To avoid a bias toward either functions or forms, we follow OPM's [26] convention of using a rather neutral word, "***Thing***", to represent a primitive type that can be interpreted as either object or process.

Emphasis on Functions ⟷ Emphasis on Forms

High coverage, Low resolution	***High-level Alternatives***	**Possible** sequences of events	**Possible** configurations of chosen *things*	**Possible** combinatorial arrangements of variable states
⇕	***Invariant Properties***	Process Model	Model of things (objects and processes)	Object Model
Low coverage, High resolution	***Low-Level Variables***	An **instance** of event sequences	The **instantiation** of an interactive phenomenon between chosen things	A **specific** configuration of interrelated objects

Figure 4-1 The space of modeling languages

Therefore, objects and processes in OPN are modeled as ***Things***. ***Things*** relate to each other by relationships to form a graph. We will illustrate a recursive data structure that utilizes ***Things***, ***Relationships*** and ***Graphs*** as the three linguistic primitives to represent arbitrary data structures.

4.2 Thing, Relationship and Graph

At the end of Chapter 3, we specified that a meta-language should be simple and Turing complete. To be simple, we need to use a small vocabulary. For representational efficiency, we need to find a highly condensed set of linguistic primitives while maintaining universal expressiveness. Examining formal languages, we found a common pattern among many of them. Languages in general need to distinguish entities in a set, have the ability to specify directed relationships, and have the data structure to represent collections. Accordingly, we defined three primitives in our language: Thing, Relationship, and Graph. In OPN, all statements are made out

of things, relationships, and graphs. In terms of the typing information, it can be stored in two ways. First, each unique instance of thing can be considered to be a special type. The distinction between things, relationships and graphs is another way to make a distinction between types. To store information about how types and other attributes are related to ***Things***, all ***Things*** have their private attributes, which are also ***Things***. The necessary expressiveness to describe systems with varying levels of details can be achieved by recursive applications of this nested data structure pattern.

- ***Thing*** is defined as the fundamental building block of any systems.
- A ***Relationship*** is a special type of ***Thing***, which defines the connection between two other ***Things***. It can be thought of as a binary operator. Syntactically, it's a ***Thing*** containing a ***RelationshipPart*** data structure.
- A ***Graph*** is a special type of ***Thing***, which is a container for ***Things*** and ***Relationships***. Syntactically, it's a ***Thing*** containing a ***GraphPart*** data structure, which can in turn be thought of as a set, or as a generic data structure to store information in OPN.

Thing represents the meta-operand in this language. ***Graph*** and ***Relationship*** are special types of ***Thing***. To manipulate these varying types of operands, one needs a set of operators that can properly utilize information encoded in each type of ***Thing***.

4.3 Operands and Operators

Thing, and *Eval* are OPN's meta-operand and meta-operator, respectively. *Thing* provides a flexible data model to encode a wide range of data structures. *Eval* provides a universal interface

to model various modes of interaction amongst *Things*. OPN utilizes this foundational operator/operand pair to construct its model of communication and computation.

4.3.1 The meta-operand: Thing

We use Backus-Naur-Form (BNF) [110],where:

```
"::=" means "is defined as"
"|"   vertical bars means "or"
"<>"  angle brackets are used to surround user-defined variables
"[]"  brackets surrounds elements that are optional.
"*"   star behind an element represents it could appear 0-n
      times
```

The data structure of Thing is defined as follows:

```
Thing ::= name = <string>;
          content = <GraphPart>|<RelationshipPart>|<string>;
          [value = <Thing>;]
```

Thing is the meta-operand in ***OPN***. Its content may be a string, a ***GraphPart***, or a ***RelationshipPart***. The choice of content type determines the ***Thing***'s type. In other words, string, ***Graph*** and ***Relationship*** can all be considered to be specialized types of ***Thing***. An instance of ***Thing*** always has a name associated with it, which identifies its uniqueness within the immediate context that contains it. The attributes of a ***Thing*** are stored in the ***value***, where the data content of ***value*** is an instance of ***Thing***. We will elaborate the content and uses of ***value*** later. Due to the generic nature of ***Thing***, we decided to use three different graphical icons to visualize their differences.

```
Graph ::= name = <string>;
          Content = <GraphPart>;
          [value = <Thing>;]

GraphPart ::= ThingCollection=[<Thing>*];
```

```
RelationshipCollection = [<Relationship>*];
```

GraphPart is a data structure contained in a ***Thing***. It contains two sets; a set of ***Things*** and a set of ***Relationships***. These two types of data structures make up the structural information about ***Graph***.

```
Relationship ::= name = <string>;
                 Content = <RelationshipPart>;
                 [value = <Thing>;]


RelationshipPart ::= Source=<Thing>,

                     Target=<Thing>;
```

RelationshipPart is also a data structure contained in a ***Thing***, where both ***source*** and ***target*** are instances of ***Things***. The third entry in ***Relationship***, ***value***, is also an instance of ***Thing***, included by default to store user-specified information about each instance of ***Relationship***. The visual representation of ***Relationship*** is an arrow.

```
value ::= Thing
```

The ***value*** field stores instance-specific information in any one of the specialized types of ***Thing***. It is the official extension mechanism to incorporate application-specific information. In addition to terminal values such as literal strings or numbers which can be stored in ***value***, it can also be a place where multiple levels of ***Graphs*** can be stored and accessed through this standard data structure.

```
string ::= [character*]
```

All numbers, names, symbols and universal resource identifiers (URIs) are eventually stored as “literal strings”. They are considered to be “terminal” because they usually require no further processing, unless explicitly specified otherwise.

4.3.2 The meta-operator: *Eval*

Definition of *Eval*:

Eval is a meta-operator, which contains computable knowledge to transform the state of its operands. It can be succinctly written in the following format:

Context(Eval) $\vdash$ ***I*** $\rightarrow$ ***O***

The above statement should be read as:

"*Given the knowledge specified in the **Context** of **Eval**, the operand **I** evaluates to **O**.*"

Thinking as a logician, one can often refer to the whole statement as a statement of a generic inference process. ***Context()*** and ***Eval*** together provide the resources and knowledge for the execution context, ***I*** is the input operand, and ***O*** is the output operand. The concept of deriving output ***O*** as an inference result requires some additional explanation. In the context of OPN, the outcome of an ***Eval*** operation, ***O***, can be one of the following ***Things***:

1. Based on the information contained in input ***I*** and the ***Context*** of the ***Eval*** operator, ***Eval*** creates one or more new instances of ***Things***, represented as ***O*** in this formal expression.
2. ***O*** represents a new condition in ***I***. ***Eval*** operated on ***I*** and changed its internal structure or content. It is optional to create a new instance of ***Thing O*** after the execution of ***Eval***.
3. ***O*** represents a new condition in ***Context***. ***Eval*** used ***I*** as an input operand and inferred new conditions about the surrounding context. It is optional to create a new instance of ***Thing O*** after the execution of ***Eval***.

4. Executing ***Eval*** could simultaneously create a new instance of ***Thing O*** as output, change the condition in ***I*** and ***Context***, or a partial combination among the above mentioned 3 possibilities.

OPN's execution kernel is implemented by following this single meta-operator approach, all computation and communication operations in OPN context are idealized as the consequences of applying ***Eval***. Adding, removing, changing values in all ***Things*** are done through some recurring application of ***Eval***s. From a programmer's viewpoint, ***Eval*** can also be viewed as the standard public function embedded in all ***Things***. Each ***Thing*** may change its own "***Eval***" behavior by storing localized computable structures or expressions to overwrite the default behavior.

4.3.3 Notations

OPN is a graphical language; therefore, it has a set of graphical symbols that denotes each type of its linguistic primitives. To provide a visual reference for what they are, Figure 4-2 is an OPN ***Graph*** annotated to indicate the type of ***Things*** they represent.

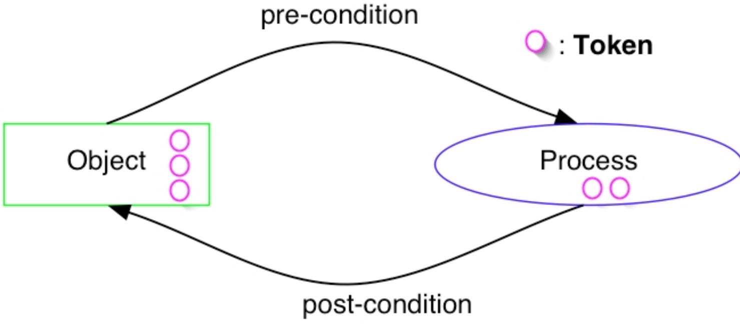

Figure 4-2 An annotated OPN

Since all data elements in OPN are represented as ***Things***, we need additional structural information to distinguish the roles of these things. Therefore, each ***Thing*** should include type information. All data elements of OPN are stored in a ***Graph***. OPN's model structure is specified as follows:

An OPN **Graph** is a ***Thing*** *that uses* ***GraphPart*** *as its content*. In the ***GraphPart*** data structure, the "ThingCollection" contains ***Objects*** and ***Processes*** as its two types of ***Things***. Similarly, the "RelationshipCollection" contains **Pre/Post Conditions** as its two types of ***Relationships***.

When a ***Thing*** *is deciated to store information about system states, it is considered to be an* ***Object****. An* ***Object*** *is a* ***Thing*** *that encodes the state of certain variables of interest. An* ***Object*** *is a passive information element in OPN. The evolutionary history of its state is recorded by a series of* ***Tokens*** *that visit the Object during execution time.* ***Objects*** *are graphically represented as a set of rectangles.*

When a ***Thing*** *is dedicated to store a set of executable instructions that will change the state of other* ***Things****, it is considered to be a* ***Process****. A* ***Process*** *is a* ***Thing*** *that represents the operators in OPN, it is an active information element in OPN. Each* ***Process*** *stores its respective operational algorithm to transform the state of certain variables of interest. These variables are associated with relevant* ***Objects****. The results of transformations are stored in* ***Tokens*** *that trigger its operation.* ***Processes*** *are graphically represented as ellipses.*

It is also possible for a ***Thing*** *to simultaneously store information about system state and executable instructions. When such situation occurs, the* ***Thing*** *is dynamically*

*determined to be a **Process** or an **Object** based on its context. The quality to become a different kind of **Thing** at different context is called polymorphism. **Objects** and **Processes** are two complementary types of **Things** that must coexist to describe communication and computation. To ensure syntactic integrity, OPN imposes a connectivity constraint between **Things**. All **Objects** and **Processes** can only connect to **Things** of the other type, but not the same type. This quality is quite pervasive in graphical representation of mathematical objects.*

*A **Token** is an instance of **Thing**.* The term "token" is borrowed from Petri Net literature [79]. It represents a communication or computation event. By default, each instance of Token contains a field "starting time", which denotes when it should be triggered to carry out the event. It is always associated with one ***Object*** that indicates where the ***Token*** is contextually situated when it is to be triggered. Once it is triggered, it serves as an input to ***Processes*** to trigger their operations and capture their operational results. As it finishes, it will be placed/associated with one ***Object***. It will be scheduled to trigger another ***Process*** when its "starting time" comes. The data content of a ***Token*** serves as the carrier of computational results and communication messages by storing a collection of ***Things*** that captures the variable states encoded in relevant ***Objects*** at the time of visit. A unique feature in OPN is that we enable all ***Tokens*** to record the trajectory information in terms of all the ***Objects***, ***Processes***, and ***Relationships*** they passed through.

*A **Pre-Condition** is a **Relationship** that specifies the relationship directed from an **Object** to a **Process**. It also stores a Boolean function that determines whether a **Token** placed in the associated **Object** should trigger the associated **Process**.*

*A **Post-Condition** is a **Relationship** that specifies the relationships directed from a **Process** to an **Object**. It also stores a Boolean function that determines whether an outgoing **Token** should be placed on the associated **Object** after the execution of the associated **Process**.*

4.4 Syntax

One of the critical tasks in architecting is to assess the global effects of local interactions. From a system viewpoint, the dependency structure of interacting variables determines how a change in one variable would affect the other variables in a system. In OPN, the users can construct a model of variable interaction by utilizing the following language constructs.

1. *OPN allows users to specify the range and resolution of variables on both the sending and receiving sides of the communication. This allows users to adaptively change the formal definition of relevant variables to accommodate the communication needs in a noisy environment.*
2. *OPN models the effects and scope of interactions by the structure of the network. The presence or absence of certain relationships between objects and processes specifies the syntactical properties of an object language specified by OPN.*
3. *Users of OPN may control variable interactions by modifying the Boolean functions in the Pre/Post Conditions.*

The key idea here is that OPN uses the graphical structure to closely resemble the structure of the problems in their original domain. The structure of the network is the syntax of the language. Specifically, the benefits of using a graph structure to represent syntactical information can be further appreciated from the viewpoint of a programming or modeling context. OPN's syntax has the following qualities:

1. *OPN's syntax is largely represented by the graph structure. It has only two types of nodes,* ***Object*** *and* ***Process****. The graph-based approach to specify syntax helps remove the syntax parsing problems in text-based languages. Every time a new character-based language is created, any minor variation in the syntax structure may introduce new exceptions to language parsing. Well-designed graphical languages, such as OPN, do not have to deal with this kind of syntax management issue.*
2. *Since OPN is designed to be a meta-language, executing OPN is about creating and modifying languages using OPN operators. OPN's simple syntactic structure simplifies the syntactic verification of its object languages.*
3. *By requiring users to specify the dependency structures between variables of interest, each user is effectively creating a localized syntax structure for their application-specific language. This makes it straightforward for OPN to function as a meta-language to manipulate application-specific models and put them into a common model repository.*

4.5 Semantics

OPN's semantic properties can be summarized as follows:

1. Higher-order Petri Net: OPN is a meta Petri Net, or a Petri Net that generates other Petri Nets [81, 82]. We store the trajectory of OPN tokens in the tokens as they run through the system.
2. The meta-data model element, ***Thing***, provides a common currency to integrate between different models of communication and computation. It is the portable and commonly known data structure that can be easily manipulated. This provides tremendous freedom and flexibility during its computation.
3. Added to each ***Process*** is a general-purpose inference algorithm, so that each process can perform localized symbolic processing to create many instances of computable expressions. It manipulates the content of Tokens in terms of ***Graph*** theoretical operators, such as adding and removing variable names. Notice that this is not a traditional hierarchical Petri Net. The Petri Net model within each subnet is another Petri Net. For practical purposes, the process model within each process is encoded as arithmetic expressions, which can be efficiently stored as a parse tree. We use a specialized parser to convert them into symbols or numerals based on available information.
4. The first two features combined created a mechanism to dynamically compose functions. It allowed the execution engine to compose computable expressions using external mathematical libraries and custom-made routines. The dynamically composed

expressions can be evaluated to produce numerical values or preserve important information that is specific to a particular simulation runtime context.

4.5.1 A layered semantic model

OPN employs a layered semantic model to manage three kinds of knowledge. The top layer is the token generating and scheduling facility, which is encoded in OPN's graphical structure. The middle layer is the token processing facility. Each instance of ***Token*** represents a distinct event in OPN's execution history. The token processing facility may be thought of as a logical inference engine, while the token generator and event scheduler can be thought of as a timing device for inference engine.

The lowest layer is the software and operating system facility, where OPN provides a variable binding mechanism to give users convenient access to third-party software libraries. OPN's user interface and language semantics are designed to hide the complexity of these low-level computing resources and represent them as variable names or functions. Chapter 5 presents how these resources are organized under OPN's implementation model.

4.6 Token Generation and Scheduling

The ***Relationships*** between ***Objects*** and ***Processes*** defines the token creation and scheduling activities. When a ***Pre-condition*** is checked and certified, it immediately creates a token to be transformed by the specified ***Process***. All ***Tokens*** move from one ***Object*** to the other through an intermediate ***Process***. A ***Process*** changes the states of a token according to the temporal and spatial context provided by its immediately neighboring ***Objects***.

After a ***Process*** finishes transforming a token, the Boolean function embedded in each of the ***Post-conditions*** connected to the ***Process*** is evaluated. If the Boolean function returns true, a new token is created. The new tokens duplicate all the information stored in the token that triggers the ***Post-conditions***. The triggering token may be discarded or stored for further analytical purposes. The new tokens are then placed onto the ***Objects*** specified by each of the ***Post-condition*** relationships.

4.6.1 The execution model of *Eval*

The ***Eval*** operators of ***OPN***'s building blocks make up its token generation and scheduling algorithm. The best way to illustrate the event generation and scheduling algorithm is to reveal how the ***Eval*** operators of different types interact with each other. At the top level, OPN has a statically defined ***Eval*** operator, whose algorithm can be defined as follows:

```
OPN.Eval
 Input(nil) # no input required
 For all Objects contained in the "ThingCollection"
     Trigger Object.Eval
```

As an intuitive way to understand how a set of statically defined ***Eval*** algorithms can create infinite recursion, imagine the OPN graph structure as a subway system, where the ***Tokens*** are the passengers in the system, and each ***Object*** is a subway station. The ***Processes*** are the trains that move passengers between two adjacent stations. ***Pre-Conditions*** and ***Post-Conditions*** are how passengers enter and leave the boarding area of each subway station. The following algorithm specifies how all possible token itineraries can be generated, assuming that ***Pre-Conditions*** and ***Post-Conditions*** evaluate to true by default.

The ***Eval*** operators for ***Object*** and ***Process*** are presented in the following pseudo code format:

```
Object.Eval
 Input(nil) # no input required
 Get the token with earliest starting time in local Token Queue
 For all Pre-conditions of this object
     Trigger Pre-Condition.Eval with the token as input
 For each Pre-Condition that evaluates to true
     Create a new token with all the data content of the token
     Get the corresponding process of this Pre-Condition
        Trigger Process.Eval with the new token as input
 Repeat Object.Eval
        Until Token Queue is empty


Process.Eval
 Input(token) # an incoming token is required
 Trigger token transforming Eval to process the incoming token
 For all Post-conditions of this process
     Trigger Post-Condition.Eval with the transformed token as input
 For each Post-Condition that evaluates to true
     Create a new token with all the data content of the token
     Get the corresponding object of this Post-Condition
     Place the token into the Token Queue of this object
```

By default, whenever no looping structures are involved, all token generation and scheduling events would come to a stop. If there is a loop in the structure, it will require a customized

condition to determine when to stop the iteration. This set of ***Eval*** algorithms provides the looping and branching features necessary to emulate a Turing Machine, thereby making OPN a Turing Complete language; therefore, we can use it to emulate any realizable algorithms, either symbolic or numeric procedures. As a model of communication, it provides a simple framework to simulate or activate the message exchanging activities. It provides a simple yet complete model of interaction from either an ***Object*** or ***Process*** perspective.

The ***Eval*** operators described above can easily include the feature to record the trajectory of each token as it moves through all the ***Objects***, ***Process*** and ***Pre/Post Conditions***. The trajectory information stored in each ***Token*** is a dynamically generated OPN ***Graph***. This ***Graph*** is not only a part of the original OPN ***Graph***, but it also computationally verifies that at least one sequence of operations can reach all the involved ***Objects*** and ***Processes***. Since all ***Tokens*** are instances of ***Things***, their content can be inspected through standard user interfaces that show the content of ***Graph***, ***Relationship***, and ***Things***.

The branching structure in OPN is graphically defined by having ***Relationships*** going outward from its connected ***Thing***. The number of outgoing ***Relationships*** determines the potential number of tokens to be generated. As mentioned earlier, each ***Relationship*** contains a Boolean expression that determines whether a new token will be generated and sent toward the corresponding target ***Object*** or ***Process***. The Figure 4-3 shows two examples.

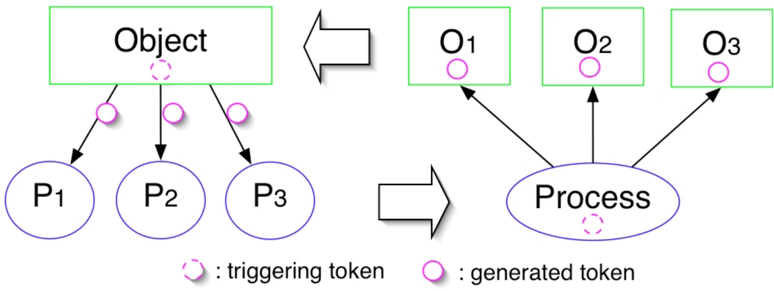

Figure 4-3 Branching and looping in OPN

Therefore, the number of unconstrained ***Relationships*** spanning out of a ***Thing*** provides a standard construct to perform token multiplication and replication. The token replication and "***Eval***" triggering mechanism is often referred to as the consumer/producer pattern in concurrent systems [111]. The branching and looping network structures coupled with the consumer/producer patterns makes OPN a Turing Complete language. The following section presents a proof.

4.6.2 Turing Completeness

This section presents a proof that shows OPN is a Turing Complete language.

Since OPN is a graph-based data structure, all token processing activities are directed by the structure of the graph. Each token's activity trajectory represents how it traverses the graph. In other words, they embody one feasible path of some graph traversal algorithm. In order to show that OPN can be used to implement any realizable algorithm, it must satisfy the condition of being Turing Complete. This section demonstrates that OPN is a Turing Complete language.

As demonstrated earlier, OPN's graphical syntax directly supports both looping and branching. These two complementary elements can be used to compose arbitrary looping and branching structures of a program. According to Böhm and Jacopini [112], looping and branching are the

two necessary and sufficient constructs to make up a Turing complete language. A simpler proof for OPN's Turing-Completeness is constructed below:

Theorem:

OPN is a Turing Complete Language

Proof:

The tape of a Turing Machine can be mapped onto a sequence of ***Objects*** and ***Processes*** connected through two relationships between each pair to form a bi-directional chain. The ***Process*** can read and write the content onto the ***Objects*** that are directly connected to it. The ***Process*** therefore emulates the read and write operations of the tape reader of the Turing Machine. The ***Objects*** serve as the different blocks on the tape to record the results of write operations. A one-to-one mapping between Turing Machine and OPN is clearly achievable. The direction of token movement is controlled by the ***Pre/Post Conditions*** that connects between the ***Objects*** and ***Processes***. The zero/one encoding of Turing Machine can be mapped onto the corresponding ***Pre/Post Conditions***. Based on this structural isomorphism, the token movements can now perfectly emulate the tape reading and writing behavior of a Turing Machine. Therefore, OPN is Turing Complete.

.

4.6.3 Model Enumeration in Finite Time

Knowing some language class is or is not Turing Complete would not normally be interesting to non-computer scientists. In the context of system architecting, it is more interesting for users to know whether an exhaustive enumeration of a finite structure can be completed within finite time and finite memory space. To show how OPN accomplishes this goal, we need to discuss how OPN creates and manipulates tokens during execution.

Knowing that cyclic structures create infinite cycles raises the question about the feasibility of exhaustively enumerating feasible models. Given a finite number of interrelated architectural alternatives, can one generate all the possible subsets of interrelated alternatives within finite time? In order to answer this question, we prove the following theorem:

Theorem:

For all static (time-invariant) and finite-sized OPN model structures, the exhaustive enumeration of all its directionally connected sub-models can be completed within finite storage space and execution time.

Proof:

We need to construct a model enumeration algorithm and prove that it would finish within finite time and only consume a finite amount of storage space. This can be accomplished in the following steps:

1. Given a static OPN model, ***G***, perform a search on its graph structure to find a set ***R*** of ***Objects*** or ***Processes*** that have no incoming ***Relationships*** (pre-conditions). Given ***G*** is finite, the search time and the resulting set, ***R*** should also be finite.
2. Create an Object ***Oi*** a Process ***Pi*** and a ***Relationship*** from ***Oi*** to ***Pi*** in ***G***. Create a set of ***Relationships*** that connect ***Oi*** and ***Pi*** to the ***Processes*** and ***Objects*** in ***R*** respectively. These additional entries changes ***G*** to ***G'***. Since all the above sets are finite, the resulting ***G'*** is finite.
3. Apply a cycle finding algorithm to find all the cyclic structures in ***G'***. Cycle finding algorithms in static and finite graphs are finite time procedures. In the worst case, the algorithm is ***O(n!)***, where ***n*** is the number of ***Objects*** and ***Processes*** in OPN. The set of cycles in ***G'*** (found by the cycle finding algorithm) will be stored in ***C***. Since ***n*** is finite, the time and storage requirement for this procedure is also finite.
4. Create an initialization token ***Ti*** in Object ***Oi***. Starting from ***Oi*** and ***Pi***, traverse ***G'*** using a breadth first approach by creating one new token for each of the outgoing ***Relationships***. As the token arrives at a new ***Object***, ***Process***, or ***Relationship***, add it to a ***Graph*** entry locally stored in the token itself. For the portion of the graph that

includes no loops, this traversal algorithm would end in finite time. All the newly generated tokens make up a set of OPN models that represent the complete enumeration of all variations of ***G'*** without the cyclic portion. This loop-less portion of model enumeration procedure would require finite time and finite storage space.

5. For ***G'*** with cyclic constructs, as the token constructs its local graph, it must check whether it contains loops as new entries are added to it. When a cyclic path emerges from the traversing token, the token would compare its locally stored graph with the set ***C***, which stores all the cycles in ***G'***. If this new path contains a segment that is already found in the set ***C***, add this path to a set ***C'***. ***C'*** denotes the set of paths in OPN that includes cyclic structures. If this path is already in set ***C'***, stop. Otherwise continue to traverse all its outgoing ***Relationships***. Knowing that the set ***C*** is finite and the number of loopless paths is finite, ***C'*** must be finite.

6. The collection of all generated tokens is the union of ***C'*** and the set of loopless paths are both finite. Knowing that all the above procedures finish within finite time and space, it proves that step 1-6 as a whole algorithm can be completed within finite time.

Having proved that enumerating OPN's directionally connected sub-models can be completed within finite time, we can make the following observations:

1. *Complete enumeration of interrelated sub-models provides a basis to create a mechanical reasoning framework to perform model decomposition*
2. *It is necessary to implement a simulation engine that would construct its sub-model as it performs computation*
3. *For graphs whose structures or parameters change during runtime, it is not possible to guarantee that full enumeration can be completed within finite amount of resources*

Clearly, finite time enumeration calculation may not be practically feasible. For problems whose computing time or storage requirements grow exponentially with their sizes, it may not be wise to conduct full enumeration. These problems are often associated with the term Non-Polynomial

complete problems, or NP hard problems. NP denotes no polynomial time or space algorithms are known. However, many real life problems are indeed NP hard, but when the problem size is small enough, even manual enumeration could solve NP hard problems. The tic-tac-toe problem is a good example. Computers can hardly have any advantage when the problem size is small. If one plays a four by four tic-tac-toe problem against a computer, the chance of the human winning is slim at best. Using a computer to expedite some of the exhaustive enumeration procedures, we can drastically expand the range of NP hard problems solvable in practice [21, 81, 25]. A computationally enabled sub-model enumeration is still highly valuable in practical settings because:

1. *Probabilistic approaches can be deployed to enumerate models with controlled parameters. They include randomized algorithms such as genetic algorithms, simulated annealing, and Monte Carlo methods.*
2. *This framework provides a measure of model complexity, so that we can use it to determine the resolution and the size of the model.*
3. *Incremental development of contextually sensitive knowledge may help reduce the size of the enumeration results.*

However, we have not discussed how to perform exhaustive enumeration for models that contain continuous variables. To resolve this problem we need to convert continuous scales into discrete categories of mathematical objects, and then enumerate the possible variations of the model. It is important to note that we are not enumerating the possible state-space configurations of a continuous model; we only intend to enumerate all the possible sub-structures of a static model. If the model is changing during enumeration time, it could alter the search space during runtime.

Therefore, it may not be possible to know whether such an enumeration program can stop in finite time. Both in practice and in theory, exhaustive enumeration is only possible when dealing with static and deterministic models. To make this enumeration approach practical, we need to find a way to convert models of alternatives into static and deterministic models. With the assistance of Group Theory and Category Theory, many classes of real world models with continuous variables and with dynamic properties can be converted into finite sized static models. Section 6.1.2 describes this approach.

Contextually sensitive knowledge is useful in eliminating unnecessary enumerations. In a simulated world, the contextual information can be emulated by computationally generated scenarios or manually created hypothetical conditions.

4.7 Token Processing

This section describes how context-sensitive information can be embedded within each generated token. All events of communication and computation are uniformly represented as ***Tokens*** carrying state information from a ***Source*** Object through a ***Token Transforming*** Process to a ***Target*** Object. A token is just an instance of ***Thing***. It inherits all the qualities of ***Thing***. We called it "token" because this is a commonly recognized term in the Petri Net literature. Like all ***Things***, a token also has a "***value***" attribute that captures the specific attributes to be manipulated through the transformation processes. A "***value***" of a token can be a primitive thing, namely a string, a number, or a computable expression, or it can also be an abstract data structure, such as a graph that contains many other kinds of things. To support automated reasoning tasks in system architecting, we need to find a way to organize numeric and symbolic calculations in a coherent

model of computation. Therefore, each ***Process*** is abstractly modeled as a generic inference engine, which not only performs numeric and symbolic calculations, but can also compile available inference rules to perform mechanical inference tasks. To provide a consistent user interface, all numeric and symbolic rules are specified in a format similar to arithmetic expressions. For example, if an input token contains the following information:

$$\textbf{\textit{Input token}} = \langle x \rightarrow 2, y \rightarrow a \rangle$$

and the ***Process*** and ***System Context*** contains the following inference rules:

$$\textbf{\textit{System Context}} = \langle g(a) \rightarrow 0*a \rangle$$

$$\textbf{\textit{Token Transforming (Process)}} = \langle x \rightarrow 3+x,\ y \rightarrow f(x),\ z \rightarrow g(y)*x \rangle$$

Then, the inferred result would be:

$$\textbf{\textit{Output token}} = \langle x \rightarrow 5,\ y \rightarrow f(5),\ z \rightarrow 0 \rangle$$

This simple arithmetic example can be visualized in Figure 4-4.

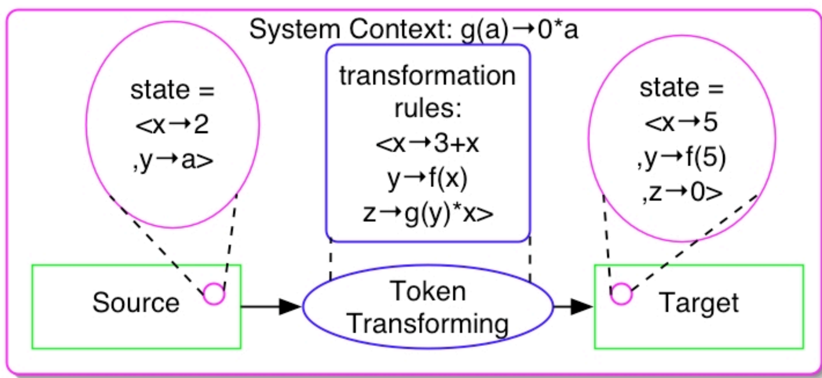

Figure 4-4 Graphical Model of Communication & Computation

In Figure 4-4, one can see the internal state of a ***Token*** is transformed based on three pieces of contextually sensitive information. As the token being transformed by the "***Token***

Transforming" process, it utilizes inference rules embedded in ***System Context***, ***Token Transforming*** process and the input token to solve for the values of all three named attributes.

We will first walk through this example visually:

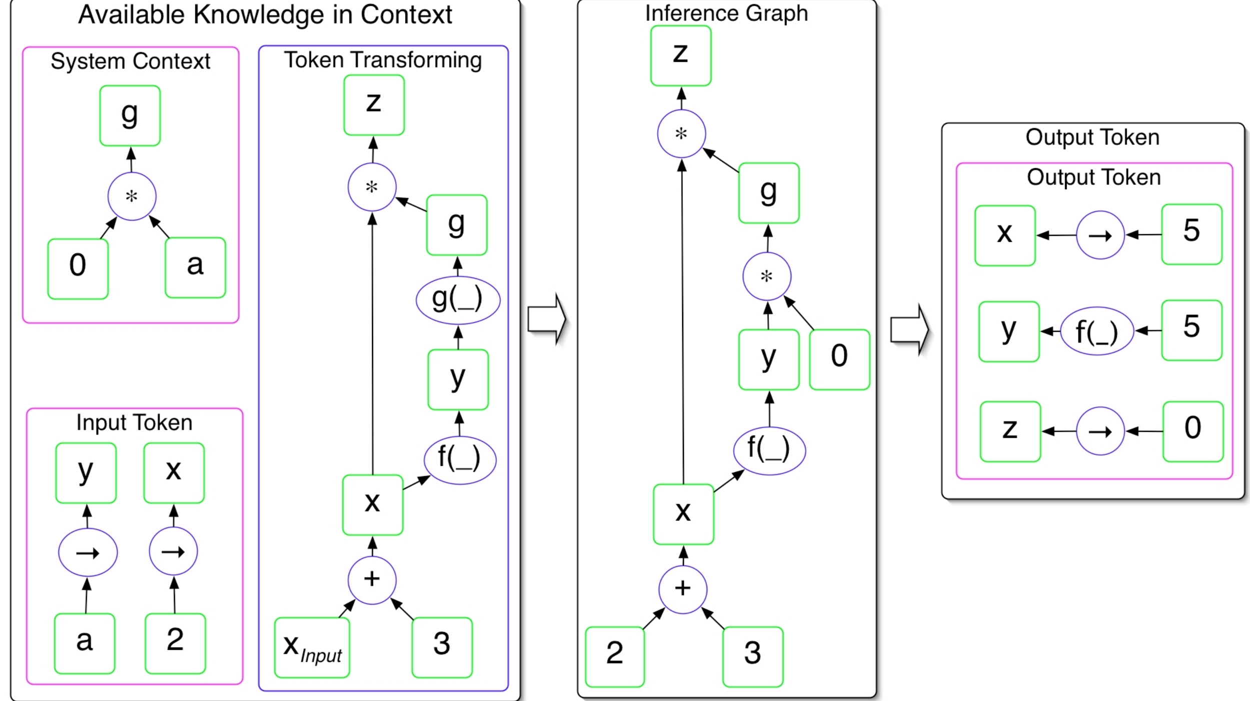

Figure 4-5 Token Creation Activities

As Figure 4-5 indicates, OPN interprets the transformation rules as executable graphs internally. All computable expressions are parsed into acyclic bi-partite graphs. In these graphs, the operands of the arithmetic formulas are treated as ***Objects*** and operators treated as ***Processes***. Once the ***Token Transforming*** process starts, it merges all three graphs into one, and performs appropriate calculations. When a numeric value cannot be found, the symbolic expression and the function signature will be stored as ***Objects*** and ***Processes*** respectively. This simple example is designed to reveal a number of important features of OPN. They are listed as follows:

1. Each ***Eval*** event triggers a series of graph-rewrite and automated inference operations

2. Context sensitive variable naming is supported.
3. A token transformation process may add new attributes to the ***Output*** token.

4.7.1.1 Context Sensitive definition of *Eval*

Let ***I*** be the token at ***Source***, ***O*** be the token that arrived at ***Target***. ***S*** and ***P*** represent ***System Context*** and ***Token Transforming*** process respectively.

The token transformation event illustrated above can be expressed in the following statement:

$$S, P \vdash I \rightarrow O$$

(The above statement should read as: "*Against the context of **S** and **P**, **I** evaluates to **O**.*")

S, ***P***, and ***I*** are three separate sources of inference knowledge to determine the resulting values stored in ***O***. The inference rules declared in each of these ***Things*** can be classified into three levels of visibility, system level, process level, and token level. As shown in the example, each of these levels may contribute zero or more inference rules to be included into the final inference graph stored in ***O***. Clearly, ***S*** and ***P*** may contain any number of inference rules. The unique feature of this inference algorithm is that token "***I***" is effectively an ***O*** of a previous process. Therefore, in a multi-staged token processing scenario, the attributes and resulting values of an output token is constructed in the steps:

Time = t-1 $S_{t-1}, P' \vdash I' \rightarrow O'$

Time = t $S_t, P \vdash O' \rightarrow O$

where

S_t represents the ***System Context's*** inference rule collection at a current time step

S_{t-1} represents the ***System Context's*** inference rule collection at a previous time step

P' represent the inference rule collection in the most recent ***Process***

I' represents the input token's inference rule collection prior to ***P'***s execution.

This equation implies that users of OPN can recursively apply this token evaluation process to dynamically compose a set of inference rules based on the trajectory of the token.

Figure 4-6 helps to illustrate this point:

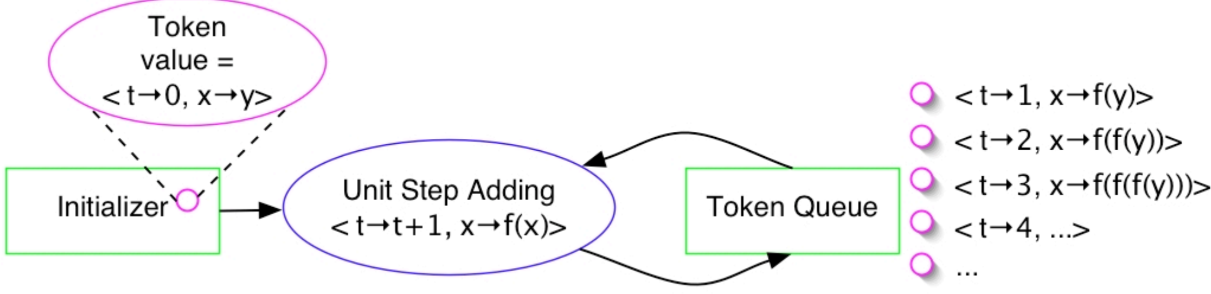

Figure 4-6 A simple recursion

Variable replacement and context sensitive rule applications are two corner stones of lambda-calculus, which is regarded as the mathematical foundation for composing computable functions. The procedures described above demonstrate that OPN provides a graphical formalism to inform users about the context of computation and allows them to specify and organize the contextually sensitive rules in an intuitive way. For complex system modeling, this computing framework offers the following utilities in model construction:

1. *Process system variables and functions as algebraic symbols. It enables users to compose a system of variables and functions that are yet defined*
2. *Provide an automated mechanism to compose inference rules*

4.8 Variable Binding

To make the inference rules perform more than simple arithmetic calculations, OPN's modeling environment must support user specified functions and global variable declaration. A bridge between the inference rules and custom-made software libraries is established to allow users to incorporate existing software libraries and define simple procedural behavior in a system simulation model. The mechanism that relates variable names with specific values or custom-defined functions is a form of variable binding. We allow users to specify variable bindings through a text file, "global script", where the values and content of globally accessible variables and algorithms are encoded. For example, the function ***f(x)*** can be defined in a "global script" written in the Python programming language as:

```
TRUE  = 1;
FALSE = 0;

def f(x):
   if x > 0:
         return TRUE;
   else:
         return FALSE;
```

Each OPN model has a unique "global script" which is "evaluated" when the "***Eval***" of the OPN model is triggered. Variables and functions defined in a "global script" are bound to appropriate referents at the beginning of this "***Eval***" operation. The variable binding mechanism serves two purposes.

1. *It defines globally recognizable variable names and algorithms in each OPN model*
2. *It serves as an interface to access software libraries beyond OPN's core library*

The first point is a standard variable scope control mechanism. It keeps rules in OPN model clean and concise. For example, if a certain number is referenced by multiple rules and must be constantly changed, it is highly desirable to create a globally accessible rule that statically defines the number with a symbolic name. Then, all the local rules that refer to this number can just use this globally defined name. It will dramatically reduce the cost of model verification and reconstruction.

The second point is to allow users to construct new algorithms using a popular scripting language of users' choice. The example shown above is a global script written in the syntax of Python programming language. The role of "global script" in OPN is to provide a standard programming interface for users to access computational resources across a wide range of software libraries. How to access software libraries is an implementation issue and will be discussed in Chapter 5.

This chapter presented the syntax and semantics of OPN. Chapter 5 describes the pragmatics of OPN, and the software architecture of our OPN implementation.

5 The software engineering aspects of OPN (pragmatics)

This section will explain our implementation strategy for OPN. We will also describe the design rationale for the user interface.

5.1 Implementation Objectives

Introducing architects to a new modeling language must not further complicate their reasoning tasks. Therefore, the software implementation of OPN addresses these concerns:

1. *How to engage users with minimal learning and configuration effort*
2. *How to leverage existing computing and communication infrastructures*
3. *How to deploy the language kernel to various platforms*

Each of these issues requires significant expertise in their respective problem domains. Fortunately, recent advancements in software and hardware technologies have made significant improvements for each of them. For example, high performance computers and network connections are not only available at affordable prices; they have also become an integral part of our lives. However, to compose a system that would simultaneously touch on all three issues is still a highly challenging technical endeavor. The combinatorial possibilities of user needs, variations in configuration management and software deployment technologies present a dauntingly large design space. To deliver a functional system within finite time, a number of architectural decisions must be made early in the design and implementation process. To paraphrase Einstein, the implementation principle is: keep everything as simple as possible, but not simpler than a Turing Machine.

5.2 The design of the language kernel

Using OPN as a system description language, we need to specify each system of interest as an executable meta-language program. The OPN simulation environment must have the ability to: represent the state-space of a system, specify system behaviors as executable programs, and provide a communication mechanism to inspect and edit the content and state of the "programs". These necessary features can be mapped onto three implementation elements: a model of the system's state-space, a view to present the internal state of the models, and a controller of the sequential actions. This Model-View-Controller (MVC) design approach originates from the Smalltalk software community [113]. It helps to decompose programming tasks to three orthogonal domains. We adopt the MVC design approach not only because it decomposes a complex implementation project into smaller and simpler chunks of implementation tasks, it also has a one-to-one mapping onto our meta-language schema. Figure 5-1 illustrates OPN's Model, View and Controller software architecture.

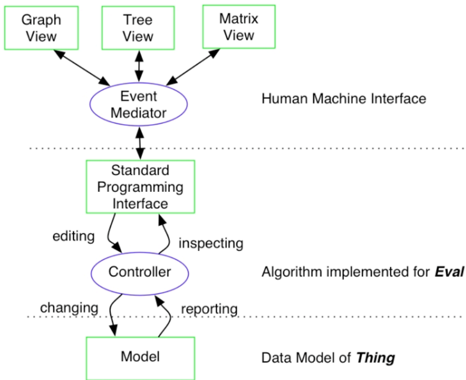

Figure 5-1 The Model View Controller of OPN

5.2.1 Model

The data structure definition of ***Thing*** (presented in Section 4.4.1) serves as the meta data "model". Through recursion, the data "model" of ***Thing*** can represent any model that can be represented as hierarchies of ***Things***. The uniformity of this meta data model makes it easy to implement one standard programming interface for ***Thing*** to manipulate and export data by people or machines. This standard programming interface becomes the protocol that mediates events between various user interface elements.

5.2.2 View

To display the structure and content of all ***Things***, each ***Thing*** in a different context may be better visualized in a different user interface component, such as graph view, matrix view, and tree view. Each "view" of the system provides a convenient conduit for people or machines to inspect or edit the data structures. Implementing a "view" in MVC is about binding an external program with the language kernel through a programming interface. All four types of activities, such as model editing, inspecting, changing, and reporting as presented in Figure 5-1, are accomplished through this standard programming interface. Using a standard programming interface makes it easier to add new views without incurring software implementation changes to the controller and model of the system [45].

5.2.3 Controller

To control the behavior of the system represented in the model we trigger the "***Eval***" meta-operator at the appropriate level of the system, to trigger all the lower level ***Eval*** operators in

context. This one controlling framework based on the meta-operator ***Eval*** is the controller of this Model-View-Controller triad.

Since OPN uses just one pair of meta-operand and meta-operator in the language specification it makes it particularly easy to implement this MVC approach. The simplicity of the language specification helps to simplify our software implementation effort.

5.3 User Interface Design

OPN is a graph-based language. We need to visualize the dependencies between ***Objects*** and ***Processes*** in a network. A screenshot is shown in Figure 5-2.

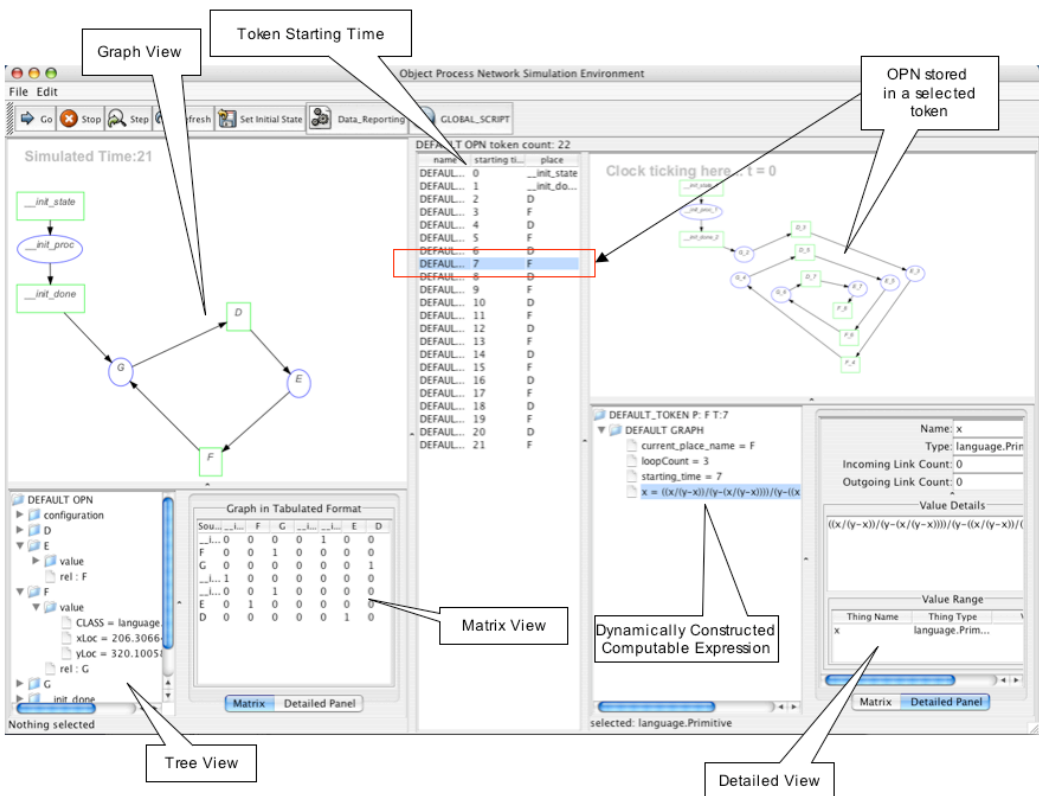

Figure 5-2 A screen shot of OPN Simulation Environment

The "graph view" and "matrix view" are provided to help users visualize the dependencies between ***Objects*** and ***Processes***. All ***Objects*** and ***Processes*** may have certain user-specified properties; we need to provide a mechanism to help users inspect and navigate the data structure and content of these properties. The "tree view" is included to enable this type of user interaction. When users are interested in viewing all the attributes of a ***Thing***, such as its unique name and its data type, a "detailed view" is provided.

5.3.1 Visualizing computationally generated OPN models

OPN is a meta-language. Its execution produces instances of "object languages". The visual interface should help users identify and visualize specific instances of object languages. As each token moves to a new "***Object***" (or "place" in Petri Net terms) in OPN, a new entry will appear in a list. The order of appearance is sorted by the token's designated starting time. This "list" is shown as a table with three columns. The first column displays the name of the token. The second column displays the starting time of the token. The third column displays the "place" (***Object***) name of the token. The "place" is the last ***Object*** the corresponding token visited. As the user selects the token by using a mouse to click on the "list", the entire itinerary of the token is displayed as an OPN model on the right-hand-side graph view. Each token stores its graph traversal itinerary as an OPN model. This mechanically generated OPN model duplicates all information stored in the respective ***Processes***, ***Objects*** and ***Pre/Post conditions*** at the time of visit and appends some sequential number to their names to make them unique. In Figure 5-2, the generated OPN model represents how the selected token moves through the "meta OPN" model. We moved around the ***Objects*** and ***Processes*** of the generated OPN model to demonstrate its

traversal itinerary. The generated OPN model is a computable model just like its "meta OPN". Each of the ***Processes***, ***Objects*** and ***Pre/Post conditions*** in the new generated model is an identical copy of the original ***Thing***.

Each token also carries attributes assigned by token processing rules embedded in the ***Processes***. These attributes are similar to the color attributes of tokens in Colored Petri Net [114]. Users can inspect each token's attributes by using the tree view and detail view below the right-hand-side graph view. The horizontal and vertical sizes of these views are dynamically adjustable. Users can determine the size of each view by dragging the handles on the dividers of these views.

The number of views necessary to perform a simulation task is dependent on the specific application scenario. Each view presented in this user interface design provides a different way to navigate around the data contained in OPN. The next section shows how users can specify inference rules and other algorithms using this user interface environment.

5.4 User Interface Framework for Layered Semantics

In chapter 4, we presented a layered semantic model to generate tokens and manipulate data in tokens. The three semantic models are token generating and scheduling, token processing, and variable binding. They will be discussed in the following sections.

5.4.1 User Interface for Token Generating and Scheduling

In this user interface, token generating and scheduling algorithms are specified through the structure of the graph. As shown in Figure 5-2, the graph view contains a loop made of four ***Things*** "G", "D", "E", and "F", and their corresponding ***Pre/Post conditions***. By default, the

"__init_state" contains a token ready to be moved over to the following ***Process*** "__init_proc". The "***Eval***" algorithm specified in Section 4.7.1 utilizes this initial condition to generate and schedule all the tokens according to the structure of the graph and rules embedded in the ***Pre/Post conditions***. In a cyclic structure, such as the one shown in Figure 5-2, the token generating sequence will not stop without additional constraints. To specify this constraint, a computable expression that returns a binary value must be made easily accessible to the user. This is accomplished by providing an "inspection" panel to display and edit the computable expression. Inspection panels are displayed by users clicking on the box, ellipse, or arrow of the corresponding ***Thing*** with the middle mouse button or by holding the "shift-key" on the keyboard while clicking with any mouse button. As shown in Figure 5-2 and 5-3, there is a ***Post condition*** between ***Process*** "G" and ***Object*** "D". Its inspection panel is the bottom "window" in Figure 5-3. The inspection panel contains a text area that allows users to view and change the computable expression. The variables in this expression can come from attributes specified in the passing token, or globally defined variables. As demonstrated in Figure 5-3, the function "f(x,MAX)" is defined in the global script inspection panel. All Things in the global scripts' residing OPN can access this function, for example, it is used by the G-D ***Post condition***. The global script inspection panel may define multiple functions and can be used as a simple interactive programming environment. Its implementation detail is discussed in Section 5-8.

5.4.2 User Interface for Token Processing

The user can specify relevant "inference rules" using a ***Process*** inspection panel as shown in Figure 5-3. (The panel on the upper left hand corner.) In this example, the inspection panel shows a computable expression:

```
x = x/(y-x)
```

This expression should be treated as a "rewrite rule". A semi-colon is used to separate rules. For example, if three rules are presented as follows:

```
z = 3 * x;          x = x/(y-x);          y = 3
```

Then the token processing routine as described in Chapter 4, will rewrite these rules as:

```
z = 3 * (x/(3-x));  x = x/(3-x);          y=3
```

This example is to illustrate that the ***Process*** inspection panel is a user interface for logic or rule-based programming. The rules written in the ***Process*** inspection panel describe intermediate stages of rule transformation. They are rules that can rewrite other rules. These rules also are applied to rules that are embedded in the passing token. The significance of this design is that OPN leverages the structure of the graph to decompose rule-based programming to the ***Process*** level. Users perform rule-based programming within individual ***Processes***. Each ***Process*** is an independent inference engine. The interactive effect of these rules embedded in different ***Processes*** is realized via tokens that pass through a series of different ***Processes*** or tokens that pass through certain ***Processes*** repeatedly. This user interface allows different domain experts to visualize the global effects of local decisions based on the structure and content of the OPN model. Different users need only focus on their area of local expertise and specify rules in the corresponding ***Processes***.

5.4.3 User interface for Variable Binding

Users need a convenient user interface component to access computational resources in the existing information infrastructure. To satisfy this need, a global script inspection panel as shown in Figure 5-3 is incorporated as part of the user interface. It allows users to define arbitrary global variable names and specify globally accessible software functions.

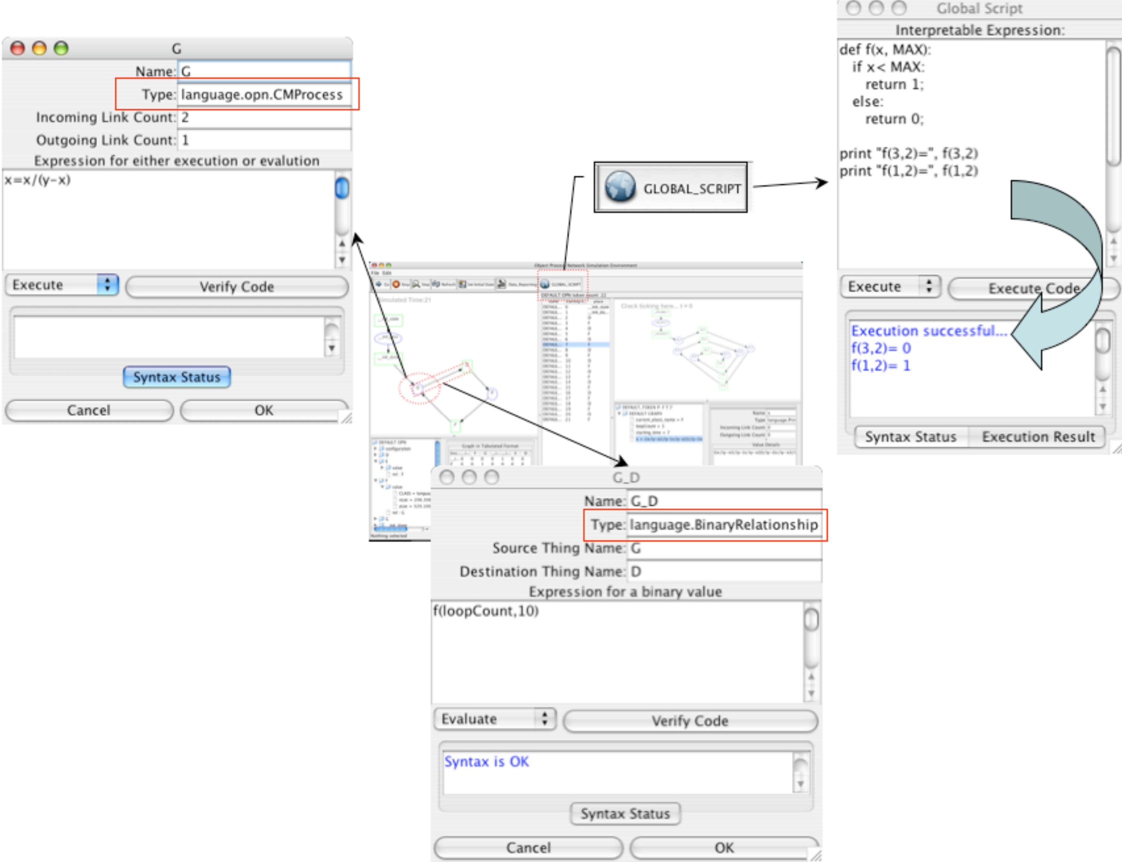

Figure 5-3 User Interface Elements for specifying inference rules/algorithms

This variable binding mechanism gives users access to user-specified algorithms, real time sensor data, database content retrieval, and legacy software libraries. Through variable binding,

we can connect the token processing events and the ***Pre/Post conditions*** (the Boolean go/no-go functions) to a variety of domain-specific software libraries that are difficult or impossible to recreate in OPN. For example, if the OPN model needs to check the real-time price data of certain stocks, this piece of information must come from an external data source. It is impossible to specify a set of rules to generate this information. We use a user-customizable global script to achieve this goal, global meaning with respect to the OPN model that hosts the script.

5.4.4 User Interface for model detail inspection

The ***Objects*** and ***Processes*** in an OPN model may contain complex data structures. Users of OPN often need to access detailed information with these data structures. As shown in Figure 5-2, when users need to inspect the numeric value of a particular attribute embedded in an ***Object***, "Tree View" and "Detailed View" provide the navigation and display facility to inspect the value. When users select a particular entry in the Tree View, the Detailed Panel (View) and Matrix View will display the information content of the selected ***Thing*** based on certain pre-defined display rules. In most cases, Detailed Panel simply shows the "name" and "value" of the selected ***Thing*** as described in Section 4.4. When an ***Object*** represents a discrete probabilistic variable, the Detailed Panel not only displays the name of the Object, it also shows the marginal probability function in a table form. As shown in Figure 5-4, when a ***Process*** denotes a discrete conditional probability function, the Detailed Panel displays the associated probabilistic measures in a tabulated form. The Graph View also embeds certain display rules that will reflect the detailed information content in ***Things***. As shown in Figure 5-4, when an ***Object*** denotes a

discrete probabilistic variable, the associated discrete states and the probabilistic measures for each state are also displayed in the Graph View.

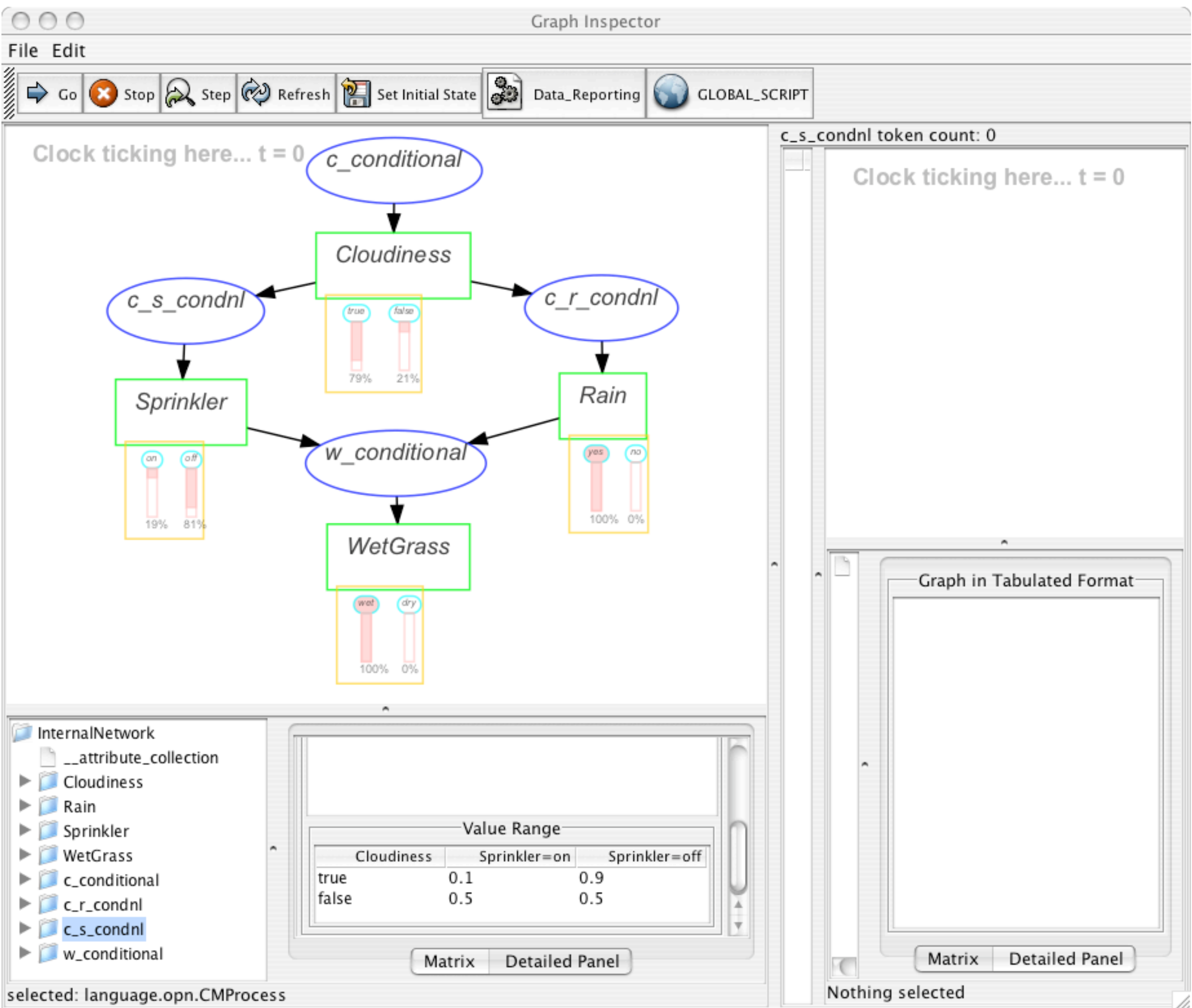

Figure 5-4 Visualizing system states in terms of probabilistic measures

The Graph View also allows users to input hypothetical "observed states" by using mouse-clicks to select and un-select "observed state" for each ***Object***. For instance, Figure 5-4 shows that two ***Objects***, "Rain" and "WetGrass", have their observed states set to one hundred percent "yes" and "wet", respectively. As users select or un-select an ***Object's*** observed state, the OPN execution engine will conduct a "belief propagation algorithm" to calculate all the relevant ***Objects'***

marginal probability values and update the Graph View accordingly. The belief propagation algorithm in OPN is based on a graphical probabilistic inference model also known as Bayesian Belief Network [66]. In theory, belief propagation algorithms can be implemented in OPN's token scheduling and processing mechanisms [69]. For performance considerations, OPN incorporates JavaBayes [115], a dedicated software library to compute the inference results. The improved performance allows architects and other stakeholders to interactively visualize the global impact of changing variable states.

5.4.4.1 Visualizing OPN with a large number of nodes

A simple interface may not be sufficiently intuitive. Human perception derives values from interactively arranging the dependency structure of a system to a particular visual orientation. That means we need to provide a storage mechanism to retain graphic layout information, such as the horizontal and vertical locations of nodes that appears on the screen. This piece of information is external to the abstract graph structure presented earlier. This additional structure must be stored in the language model in a non-intrusive way. Otherwise, every time certain changes are made to the graph the visual aspect of the information would either completely disappear or require significant computing time to reconcile the differences. To accomplish this data structure need, we simply utilized the "value" attribute of every ***Thing*** to store additional information about physical layout or other kind of information. The name for each additional piece of information is unique within the scope of the "value" attribute. When multiple attributes are stored in "value", the value of ***Thing*** turns into a ***Graph***. In other words, it stores the attributes as a set of ***Things*** in the ***GraphPart*** data structure of a ***Thing***.

To improve users' interactive experience with the graph editor and allow different sizes of graphs to be visualized intuitively, we incorporated the graphics library "Picollo", designed and implemented by the Human Computer Interface Laboratory at University of Maryland. "Picollo" is a Java implementation of a "zoomable user interface" [116]. Users can easily drag their mouse to zoom and pan the diagram to inspect the structure and perform data structure edits on the graphs visually. The graphics performance of this library is better than many competing open source and commercial products. Without this software technology, a visual programming environment would not be cognitively appealing to human users.

5.5 Enabling Technologies

To attain portability across a large number of platforms and reduce configuration effort, Java was chosen as the primary implementation language. This choice enables the OPN simulation environment to execute on any computers that run the Java 2 Platform Standard Edition (J2SE). All algorithms specified in the earlier chapters are implemented and statically compiled using the Java programming language. For model storage and other communication related functions all instances of OPN models are stored in an industry standard format called Extensible Markup Language (XML). This textual format of data encoding allows one to send OPN models as a stream of characters over the network so that different simulation models and simulation results could be shared and distributed across multiple locations. For users who desire instant feedback from the simulation environment, we incorporate Jython [117], a Python language interpreter implemented in Java, to provide an interactive programming mode so that users can issue any

Java function calls without having to compile and link the intermediate object code (bytecode) [108].

5.5.1.1 Layered Software Architecture

The overall software implementation strategy can be visualized as a layered architecture. Starting from the bottom, the source code of OPN is divided into three packages: OPN Persistence, OPN Language Core, and OPN User Interface. The overall code structure is presented in the following block diagram.

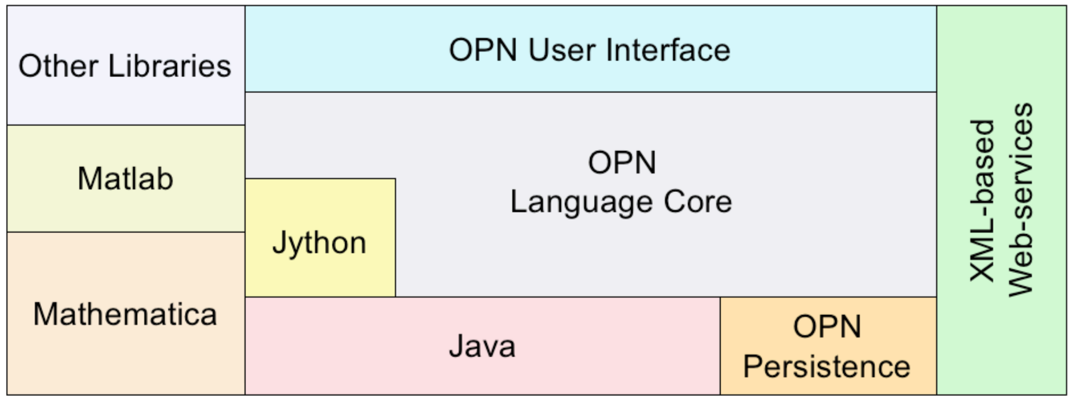

Figure 5-5 Software components

OPN Persistence is a set of bidirectional data storage and extraction algorithms written in Java that translate between OPN models stored in standard XML form and an in-memory data structure that can be interpreted by algorithms specified in OPN's Language Core. The standard XML form also enables communication with other software applications that read and write data in standard XML form. The Language Core package contains the data structure definition, token scheduling/creation and token processing algorithms that are all implemented in the OPN language core. Variable binding and algorithms that require responses from remote computational services such as Mathematica's or Matlab's kernel can be accessed through

Jython or specialized adaptors implemented in Java. The OPN Language Core does not have any dependency on the OPN User Interface package. This independence allows the Language Core package to be deployed to computing environments that do not require user interfaces. The OPN User Interface package contains the user event management routines that allow users to navigate around OPN models stored in local memory. The software architecture is designed in a way to enable any instance of OPN to be inspected by User Interface programs running on remote computers.

5.5.1.2 Language Core and Models of Computation

On a digital computer, all models of computation are emulated through discrete events. To provide a maximum level of flexibility in representing real world systems, the events in OPN's are assumed to be asynchronous. In other words, events that are being executed should not force later events to wait for them to finish. This was easily accomplished using Java's threading features. However, when the language needs to be implemented on other languages that do not support threading, the asynchronous issue must be explicitly addressed.

The key reason for us to choose Java as an implementation language is its popularity. A wide variety of open source libraries are available. For instance, to perform probabilistic inference calculation we incorporated JavaBayes [115], an open source library that includes efficient algorithms for solving probabilistic network inference problems. Another key element of our language model is the ability to interpret Python expressions through a Java library called Jython [117]. Jython gives us an expression interpreter to make Java function calls during model simulation time.

The combination of Java and Jython gives us a number of powerful libraries to perform string manipulation. The token processing mechanism is implemented in Java. For user specified functions we use Jython as the programming interface to encode procedural code specification. Jython allows users to directly call any functions implemented in Java. If a complex software function is implemented in languages other than Java, it is still technically feasible to perform function calls through Java's Native Interface.

This chapter presented how OPN was implemented. Chapter 6 presents how OPN can be applied to large-scale socio-technical projects such as the Apollo Program.

6 Case Studies

This chapter provides three case studies on applying OPN to reason about architectural decisions; they are:

- The Apollo Program (retrospective)
- NASA's Space Exploration Initiative (current)
- Enhanced-Ground Testing Pod

6.1 The Apollo Program (retrospective)

President Kennedy's historical remarks best explain our choice of performing a retrospective case study on the Apollo Program using OPN:

> *"We choose to go to the moon. We choose to go to the moon in this decade and do the other things, not because they are easy, but because they are hard, because that goal will serve to organize and measure the best of our energies and skills, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win, and the others, too."*
>
> J.F.K. September 12, 1962

To paraphrase JFK's language of persuasion, we choose Apollo as the benchmark project for architectural reasoning because it is still hard. It is still expensive and complex. It embodies all the qualities of a large scale socio-technical system. We choose Apollo to demonstrate that architects must be able to utilize a domain independent language to organize resources and measure complexity in their respective projects. A well-executed program such as Apollo must

seamlessly connect many human organizations and technical disciplines. Similarly a well-designed meta-language should serve as a medium to connect organizations and disciplines by enabling seamless exchange of knowledge. We will demonstrate that the concept, theory, and tools for manipulating meta-language can help architects formally describe spaces of alternatives, generate alternatives, and calculate performance metrics. The goal is to demonstrate that meta-language as an instrument for reasoning is applicable to a wide range of socio-technical challenges such as Apollo and the others, too.

6.1.1 Where an executable meta-language is applicable

To illustrate that an executable meta-language framework is applicable to large-scale projects like Apollo, a series of executable models expressed as OPN object languages are developed using an OPN model as the meta-language. Three instances of object languages are defined. ***Mission-Space*** utilizes the declarative language features of OPN to specify the space of mission modes. ***Mission-Enum*** utilizes the imperative language features of OPN to generate individual mission modes. ***Metric-Calc*** utilizes the simulation language features of OPN to perform metric calculation.

6.1.1.1 Specify the space of mission alternatives

Kennedy's declarative language defined the game, "Reaching the Moon first", as a common language for two nations of people[1]. To mobilize a wide range of stakeholders and organizations to participate in his game plan, Kennedy used a declarative statement to enable efficient

[1] November, 1989, three Americans, Kerrebrock, Young, and Crawley [Korolev, p.306], took pictures of the N-1 program's remaining equipment at the museum of Moscow Aviation Institute which confirmed the existence of the former Soviet Moon-bound program.

communication and justify resource allocation. On the highest level, Kennedy articulated a goal in a political language that robustly justified the value of the proposed program across multiple system levels and in a global context. Kennedy implicitly specified the space of mission alternatives by explicitly stating that Americans must go to the Moon within a decade.

6.1.1.2 Enumerate mission alternatives

Architects also need a mechanism to create individual instances of mission alternatives to compare and contrast the pros and cons among them before a mission architectural decision can be made. A declarative language can only specify what to do; it doesn't provide the how.

Brainerd Holmes, Apollo's program manager, presented the following statements before the House Committee on Science and Astronautics, one day after NASA internally selected the Lunar Orbit Rendezvous mission mode from other contending alternatives [118, 119]:

> *"It was quite apparent last fall this mission mode really had not been studied in enough depth to commit the tremendous resources involved, financial and technical, for the periods involved, without making ... detailed system engineering studies to a much greater extent than had been possible previously. ... but investigation could go on forever, ..., at some point one must make a decision and say now we go…"*

Holme's argument clearly indicated that a comprehensive study is desirable but not affordable, even considering the tremendous risk and consequences involved in a politically important project. However, the role of an architect for a large socio-techno system is to avoid unnecessary risk by making informed decisions. If any critical decisions must be changed at a later time, the entire project could fail. The ability to reason about critical decisions with incomplete information is an inherent challenge in most architecting process.

In the context of the Apollo Program, the LOR decision was announced in 1962. This decision was highly controversial. To visualize the gap of available knowledge between the point of decision and the point of realization, NASA's budget allocation for Apollo serves as a good reference. The budget allocation over time is visualized in Figure 6-1.

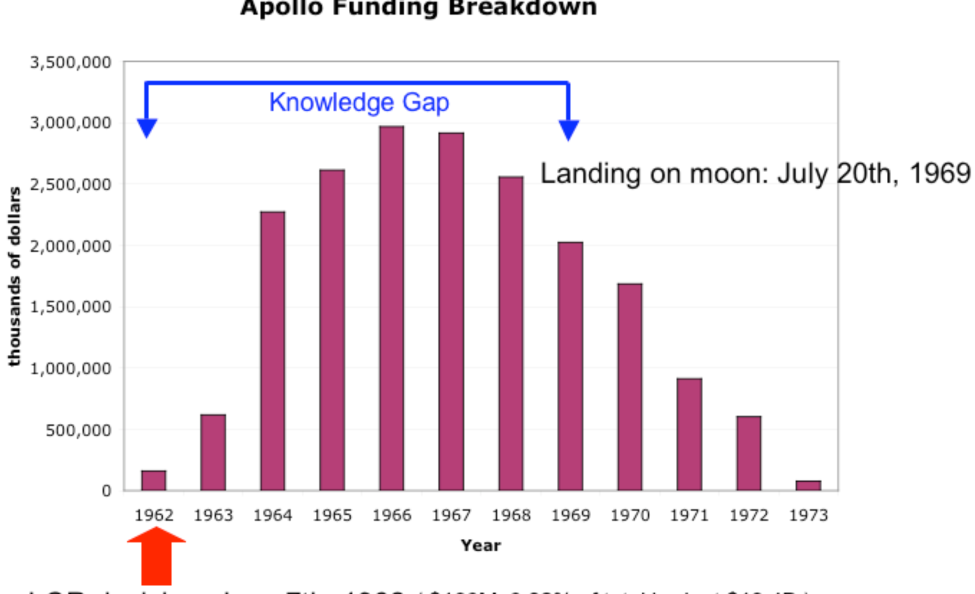

Figure 6-1 Apollo Funding Breakdown

Figure 6-1 shows that at the point of LOR decision, the Apollo Program was only about one year old and the total amount of allocated budget was $160 million. This is less than 1 percent of the $19 billion dollar program. This historical evidence demonstrates that architecture decisions are often made at a time when limited knowledge is available, and significant risk and uncertainty are inevitable.

6.1.1.3 Making tradeoff decisions about mission architectures

The Apollo program can be formulated as a constraint satisfaction problem: how to get to the moon and back by the deadline given finite resources. In this problem, all three elements can be

expressed in definite terms. We know where the moon is, the deadline for touch down is known, and the budget is not infinite. However, the number of possible configurations for the program is infinite. For an experienced program manager, to reduce complexity of this combinatorial problem, the essential variables must be identified and contained. Apollo's organizational structures, technology development tasks, physical devices, and geographical locations of various teams evolved around one central theme, the high level trajectory and operational sequence of the spacecrafts. It was so important in the program that an official term was assigned to it: "***mission mode***". Any slight alteration in the mission mode may trigger costly changes across the entire program. Therefore, the chosen mission mode is the common protocol that defines the organizational interfaces and design activities among various operational and technology development teams. If the mission mode were fatally flawed, the consequences would be unthinkable. The mission mode, in this context, defines the base architecture of the program. As indicated earlier, architects and other stakeholders are extremely interested in rigorously reasoning through the space of trajectory alternatives.

6.1.2 Specifying mission architectures in formal languages

To reason about Apollo's architectural alternatives, we need a declarative model to comprehensively specify the space of possible trajectories in terms of where and how the vehicles move between the Earth and the Moon. To generate all trajectory instances, an imperative model is needed to specify an efficient algorithm and generate the complete set of trajectories. Finally, we need a simulation model to assess performance metrics for each of the generated trajectories. Each of these three models can be considered as an instance of language.

All these instances of language can be created using Object-Process Network's visual language construction environment. We develop the three functional languages in successive steps. They are identified as:

1. ***Mission-Space***: the declarative language that specifies the space of trajectory alternatives
2. ***Mission-Enum***: the imperative language that generates trajectory instances
3. ***Metric-Calc***: the simulation language that calculates the metrics to compare and rank the competing mission architectures

Describing these abstract languages successively has two advantages. First, we can define a domain-specific vocabulary to discuss the formal properties in each area of the architectural reasoning task. Second, we can utilize the concepts and structures developed in an earlier model to support functional requirements in related reasoning tasks. In other words, these object languages are meta-languages themselves. They can be used as a basic language structure and incrementally evolve into domain-specific languages to better utilize knowledge in various contexts. These three languages also extensively utilize the layered semantic model, as described in Section 4.5.1.

6.1.2.1 Issues related to representational economy

In a continuous space, the number of trajectories between two spatial locations is infinite. It would be infeasible to comprehensively enumerate all possible trajectory models if continuous parameters are involved in the enumeration. Based on the theorem proven in Section 4.7.3, we know that as long as the meta-model is discrete and finite, we can enumerate all possible sub-models in finite time. To accomplish this conversion, we need a finite set of operands and operators that can comprehensively describe the dynamics of the system. Group theory [120]

provides a mathematical foundation to convert formulas that describe continuous spaces into a finite set of operators and operands. Frazolli [121] utilized these mathematical methods to formulate a modeling technique that "quantizes" the description of continuous dynamic systems into a finite set of motion primitives. Frazolli's idea is illustrated in the following diagrams:

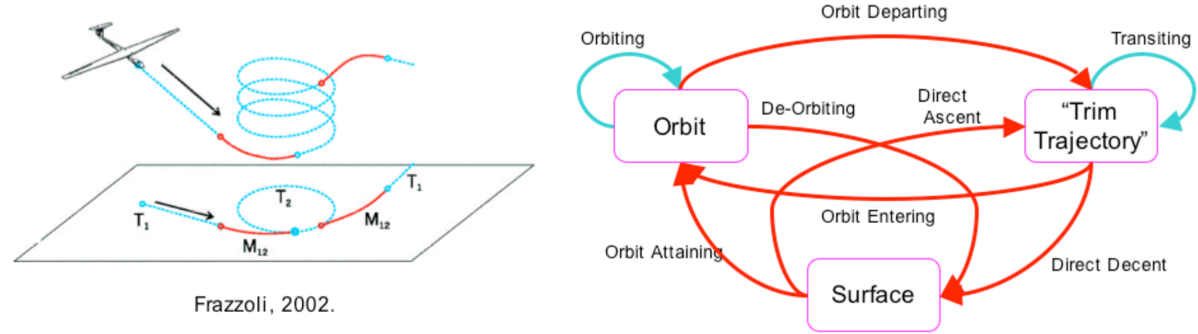

Figure 6-2 A continuous space quantized in discrete vocabulary

By dissecting the motion of an aircraft or any object into two classes of motion primitives, namely repeatable and finite time motions, Frazolli created a simple language that uses two kinds of linguistic primitives to describe the space of all possible continuous trajectories. As shown in the figure above, the two kinds of motions are shown in different colors. Motions at constant speeds or constant accelerations are classified as repeatable motions, such as "Trim", "Surface", and "Orbit". All other (non-constant) motion speeds and accelerations are considered to be finite time (transient) motions, such as "Direct Decent", "Orbit Attaining", and "Orbit Departing". This elegant formulation was applied by Frazzoli to build modeling tools for motion planning of autonomous vehicles. To employ these theoretical techniques requires sophisticated mathematical knowledge, which may not be intuitively communicable to non-technical stakeholders in a program like Apollo. Therefore, a meta-language needs to play the bridging

role to hide the technical complexity and communicate the logical ideas to a wide range of stakeholders.

Using a graphical meta-language, such as OPN, the motion primitive idea can be mapped onto a bi-partite graph. In the context of the Apollo program, Frazolli's language of quantized motion primitives can be used to model all possible mission modes in OPN.

6.1.2.2 Representing Apollo's space of architectural alternatives

Having illustrated the theoretical aspect of the modeling vocabulary, a domain specific vocabulary must be injected to make this system useful. Given the context of the Apollo program, a spacecraft's possible trajectories can be modeled in the following language: ***Mission-Space***.

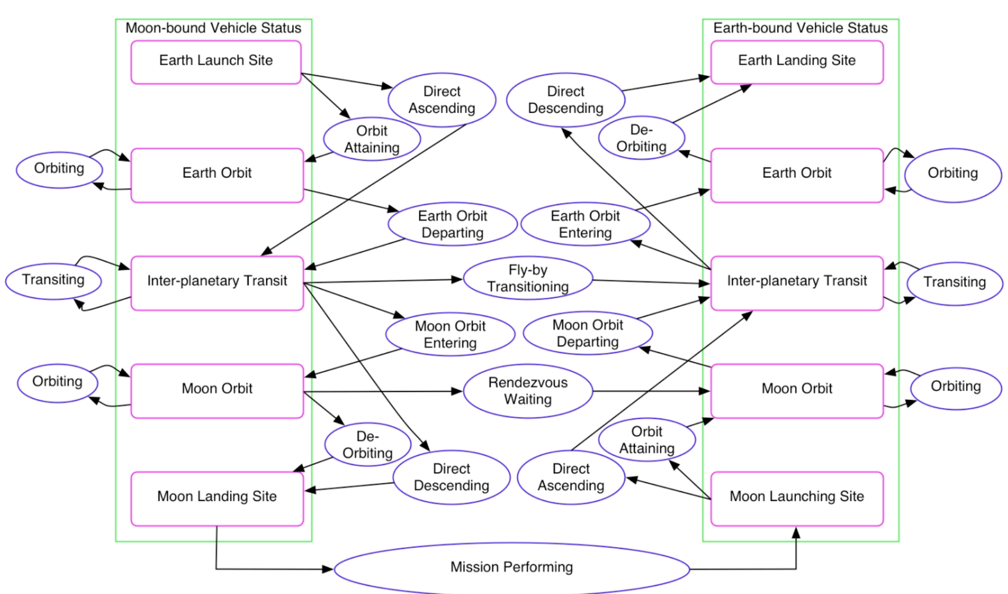

Figure 6-3 Specialized Vocabulary for the Apollo Program

Mission-Space is an OPN model that utilizes Apollo-specific vocabulary to describe the space of mission mode alternatives. We associate repeatable motions with the ***Objects***, shown as the rectangles and finite motions with the ***Processes***, shown as the ellipses. The following textual description illustrates the nomenclature in detail:

Mission-Space is an instance of ***Thing***. In its ***ThingCollection***, it contains ***Finite_Time_Motion*** and ***Repeatable_Motion*** as its two types of ***Things***.

Mission-Space ::= ThingCollection, RelationshipCollection, value

ThingCollection ::= Finite_Time_Motion, Repeatable_Time_Motion

RelationshipCollection ::= Pre-Condition*, Post-Condition*

value ::= starting_location

where

Repeatable_Motion *is a set of* ***Objects*** *that represents all the repeatable motions in the language of mission modes. To make the language easily readable, we use the "location" of the vehicle to denote a specific instance of motion primitive. For example, "Earth Launch Site" is used as the name for the first repeatable motion because the vehicle is at rest where its speed and acceleration are both zero. The other instances of* ***Objects*** *are "Earth Orbit", "Moon Orbit", and so on.*

Finite_Time_Motion *is a set of* ***Processes*** *that represents all the finite time motions (or maneuver operations) in the language of mission modes. For each maneuver operation, a matching* ***Process*** *is named by a verb in a gerund form to denote a specific operational task. For instance, "Direct Ascending" denotes the vehicle moving from the "Earth Launch Site" into "Inter-planetary Transit". During this transitional phase, the vehicle is moved through a series of complicated acceleration and deceleration stages. The vehicles' speed and acceleration profiles are constantly changing, therefore, making it a finite time motion. The other finite time motions that*

were considered by the Apollo program office are represented as ***Processes*** *respectively.*

Pre-Condition *is a set of arrows that specifies the acceptable transitions between repeatable motions (**Objects**) and finite-time motions (**Processes**). Each arrow represents a binary conditional statement that determines whether a vehicle can or cannot be transitioned from a specific repeatable motion state into a specific finite-time motion. By default, the conditional statements all evaluate to true. Users can customize these conditional-statements based on domain-knowledge.*

Post-Condition *is a set of arrows that specifies the acceptable transitions between finite time motions (**Processes**) and repeatable motions (**Objects**). As the complementary knowledge of **Pre-Condition**, it checks whether the finite time motion can be stabilized into a repeatable time motion. By default, the conditional statements all evaluate to true. Users can customize these conditional-statements based on domain-knowledge.*

value *is a **Thing** that specifies the vehicle's starting location. It specifies the boundary/initial condition of each trajectory alternative space. For the Apollo program, the starting-location is evidently the "Earth Launch Site".*

6.1.2.3 *Mission-Space* is a declarative language

This description of alternative space provides the formal syntax and semantics to describe all possible instances of architectural alternatives. The ***Objects*** and ***Processes*** in the above mentioned models serve as the nouns and verbs of the mission mode language. Instead of giving each mission mode a specific name, each mission mode is a unique composition of motion

primitives. The graph structures of these models are analogous to the grammatical rules of these languages. The arrow directs the possible sequential transitions between two primitives. In the classification scheme of languages, these descriptive languages belong to a class of languages called "declarative language". Declarative languages only specify **what** can be said without providing imperative instructions on **how** to create individual instances of them.

6.1.3 Generate all possible trajectories

The models presented above only serve as declarative specification of architectural alternatives. It only shapes the space of alternatives without providing a mechanism to produce concrete instances. In practice, architects need efficient algorithms to generate concrete instances or distinctive classes of architectural alternatives, so that they can further investigate the qualities of varying architectural alternatives. An imperative language is needed to specify the sequences of actions that computing devices could follow, to generate architectural instances sequentially. Using the motion quantization methods, we have compressed the space of continuous motion into a language based on discrete representational symbols, we now need to introduce imperative semantics to the language. An imperative language must allow its users to specify the execution order of its instruction sets.

6.1.3.1 An imperative language extension to *Mission-Space*

To enumerate all possible mission modes, we need to specify an algorithmic model to construct all the possible mission modes. ***Mission-Enum*** is created to provide imperative language features to the declarative language ***Mission-Space***. Based on information already encoded in ***Mission-Space***, we can utilize the graph structure to control the direction of alternative space

exploration. Therefore, ***Mission-Enum*** extends ***Mission-Space*** by adding a generic execution algorithm that is applicable to all "programs" written in the language ***Mission-Space***. The algorithm can be specified as follows:

```
Mission-Enum.Eval
  Input(Mission-Space, destination Object)
  Get the "starting_location" Object from Mission-Space
  Create a token that represents a vehicle
  Set the starting_time of the token to "0"
  Place the token into the Token Queue of the "starting_location" Object
  Trigger the Eval operator of the "starting_location" Object
  While at least one Object's Token Queue is not empty
        Wait
If all Objects Token Queues are empty
        Report the History List the specified destination Object
```

In this language, all the Objects and Processes implement the same ***Eval*** algorithm as specified here.

```
Object.Eval # All Objects are instances of Repeatable Motion
  Input(nil) # no input required
  Get the token with earliest starting time in local Token Queue
  For all Pre-conditions of this object
      Trigger Pre-Condition.Eval with the token as input
  For each Pre-Condition that evaluates to true
      Create a new token
      Copy all the data content of the selected token to new token
```

```
        Get the corresponding process of this Pre-Condition
          Set the starting_time of the new token based on current time
          Trigger Process.Eval(new token) # new token as the input
     Repeat Object.Eval
          Until Token Queue is empty
```

The algorithms for Process's ***Eval*** is specified here.

```
    Process.Eval # All Processes are some instances of finite time motion
     Input(token) # an incoming token is required
     Construct or modify the token's trajectory information by:
          Add the following Things to the incoming token's trajectory
               The token's originating Object, and the Pre-Condition
     Add the duration of processing time to the token's starting_time
     For all Post-conditions of this process
        Trigger Post-Condition.Eval with the transformed token as input
     For each Post-Condition that evaluates to true
        Duplicate transformed token data into the newly created token
        Get the corresponding object of this Post-Condition
        Construct or modify the new token's trajectory information by:
          Add the following Things to the token's trajectory
               The current Process, and the Post-Condition
        Place the token into the Token Queue of this Object
        Place the same token into the History List of this Object
```

Executing the ***Eval*** algorithm of ***Mission-Enum***, returns a complete set of feasible mission modes.

This approach has the following properties:

1. It visually and sequentially resembles how spacecrafts travel through space, therefore making it easy to intuitively verify the dynamics of the enumeration algorithm.
2. All the tokens placed in varying ***Objects'*** History List by evaluated ***Processes*** are verified by all the constraints specified in the ***Mission-Space*** language. All generated tokens in the History Lists are considered to be feasible by definition.
3. Each token records its trajectory information in terms of all the primitive motions and ***Pre/Post-conditions*** it visited. Therefore, once the ***Mission-Enum*** algorithm is executed, it not only creates a report that contains a set of possible mission modes. Each mission mode report is a computable model by itself. One can think of ***Mission-Enum*** as a generic enumeration engine for graph-based simulation models.

6.1.3.2 Computational Results of *Mission-Enum*

The following screen shot illustrates the user interface and one instance of the mission mode automatically generated by the ***Mission-Enum*** language.

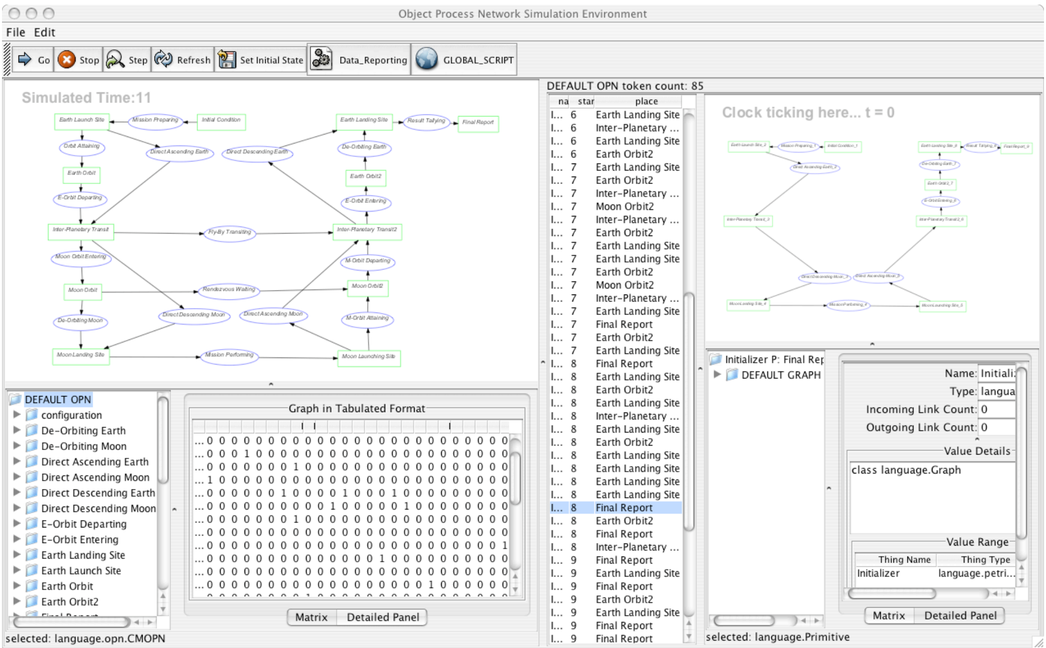

Figure 6-4 A Screen Shot of the OPN Simulation Environment

This interface displays the original model that specifies the space of mission alternatives on the upper left hand corner. The table in the middle lists all the generated mission modes. As a user selects one of them, a mission mode is displayed in graphic form on the upper right hand corner. Each mission mode is stored in a token as an OPN model. The OPN model can be extracted and stored as a separate model to perform more focused analysis on that mission mode. This interface provides an interactive environment for architects and other stakeholders to visualize mission mode variations.

6.1.4 Calculating performance metrics

Metrics of architecture are necessary means to compare the goodness of architectural alternatives. To arrive at a decision, a preference order must be established between all available mission modes. The meta-language should provide the means to compute performance metrics, otherwise,

a large set of unstructured lists of alternatives would provide little or no value to decision makers. In the case of the Apollo Program, the key driver that ultimately determined the mission mode was the size of the rocket [122]. The rocket, Saturn V, which was ultimately developed in the Apollo Program, is still the largest rocket ever built. This high technical achievement also signals high risk. Finding mission modes that would reduce the required rocket size would have been of great interest at the point of decision.

6.1.4.1 Calculating performance metrics using Metric-Calc

Mission-Enum gives us the ability to generate instances of mission modes. We now need to perform user-specified metric calculation for each of the generated mission modes. In this case, we would be interested in calculating the total mass and the probability of operational success of each mission mode. The new language should be able to support an arbitrary number of metrics as needed. As demonstrated earlier, ***Mission-Enum***, only specifies the algorithm for generating missing mode instances, but does not have an explicit mechanism to incorporate detailed numeric or symbolic reasoning. To incorporate this new feature to ***Mission-Enum***, we add these metric-calculation language features to it and call this new language: ***Metric-Calc***. There are many advantages to directly embed metric calculation features in a language similar to ***Mission-Enum***; it provides a programmable interface to eliminate the unnecessary enumeration tasks based on calculated metric values. For example, if a certain mission presents an unacceptably low probability of success, a corresponding ***Pre/Post-Condition*** should detect the case and determine whether to generate mission modes according to a user specified Boolean expression.

6.1.4.2 *Metric-Calc* as a simulation language

The ability to perform scenario-based metric calculation is a key feature of simulation modeling languages. Simulation modeling languages provide a computational instrument for architects to observe certain performance metrics given a set of what-if scenarios. In the field of space mission design, metric calculation routines can be quite complex. Some of them require iterative procedures that must be implemented in high precision algorithms and/or high performance programming languages. To accommodate performance metric calculation features, ***Metric-Calc*** must provide a convenient interface for users to specify either numeric or symbolic expressions.

Therefore, the ***Metric-Calc*** language extends ***Mission-Enum*** by including an inference engine that can handle arithmetic equations and incorporate results from procedural algorithms. In the OPN modeling environment, these two features are provided through specifying rules in ***Processes*** and binding variables declared in "Global Script". Two levels of programming interfaces are specified. All the ***Objects*** and ***Processes*** in the same language instance have access to variables, inference rules, and procedural algorithms defined in the "Global Script". On the ***Process*** and ***Pre/Post Condition*** (Relationship) levels, all rules are specified in terms of Boolean expressions or arithmetic formulas. These rules differ from the declared variables and rules in the "*Global Script*". First, they are not visible to other ***Processes*** or ***Pre/Post Conditions***. Therefore, it allows users to control and isolate unnecessary interactions between variables and rules. Second, these localized rules are converted into inference graphs as explained in Section 4.7. Each token processing event creates a different inference graph based on a different execution context. The dynamically generated inference graph is literally a new program customized to solve an application specific problem. When the program does not have all the required numeric

values to return a numeric answer, it retains all the accumulated knowledge in a graph structure. If there are certain variables defined by functions that are yet to be defined, it simply treats the function as a symbolic variable. Whenever the numeric values or function definition are supplied, it will be incorporated into the inference graph. As the token moves around the OPN, it accumulates more knowledge in different contexts, and enriches the reasoning power of the inference graph incrementally.

This incremental knowledge accumulation technique is one way to realize the concept developed in Domain Theory [123] and the Information-Gap Decision theory [14]. It provides an algebraic construct to temporarily store the "uncertain" factors in the reasoning process. As more knowledge becomes available, it will substitute the symbolic placeholder with more specific numeric or symbolic values. Since functions in the OPN modeling environment can be custom defined, it can be a probability distribution function, fuzzy membership function or a deterministic calculation routine.

6.1.4.3 Automatically compose metric calculation formulae

Another key issue relates to the complexity of formula construction. As ***Mission-Enum*** generates complex mission modes, the metric calculation routine may have been different for each of the mission modes. We need to formulate a model of computation so that we can rely on the information embedded in all specified language models to infer the proper composition of metric calculation formulae. In other words, we need to create a language that can automatically construct metric calculation formulas based on the context of each mission mode.

To demonstrate automatic composition of formulas, calculating total mass of the vehicle at launch time serves as an example. The formula that calculates the required fuel mass based on a

given payload and specific impulse for the fuel is called the Rocket Equation. It takes the following form:

$$e^{\frac{dV}{g \times Isp}} = \frac{M_{Total}}{M_{struct} + M_{payload}}$$

where:

dV : difference in velocity over the entire period of maneuver (ΔV)
g: gravitational constant
Isp: specific impulse of the fuel employed
M_{Total}: Total mass
M_{struct}: Structural mass
$M_{payload}$: Payload mass

The rocket equation is often solved backward using a desired payload mass to infer the total mass at launch time. This backward calculation process must accommodate variations in structural mass and specific impulse of the chosen fuel. These two values may change due to varying spacecraft configurations and fuel choice. Since each mission mode representing a different sequence of ***Finite_Time_Motion*** often implies a different spacecraft configuration, the rocket equation takes on a slightly different form for each mission mode. Due to these variations, the calculation of initial mass must be manually formulated and programmed for each mission mode.

This highly simplified composition of the rocket equation already shows signs of complexity. This equation could become even more complicated when multiple fuel types and ratios between the structure mass and propellant mass change. This thesis used the numeric assumptions presented in Houbolt's report [124] to perform vehicle weight calculation.

To calculate probability of mission success, we treat the model as a Markov Network [66]. The probability measures associated with each ***Process*** can be stated in either numeric or symbolic terms. For the purpose of illustration, we used the following numeric assumption. This numeric assumption is based on an in-person interview with Dr. Robert Seamans [125]. The focus is not about the exact numeric values, but the relative levels of risk considerations.

Reference Risk		**Higher than reference risk**	
To earth orbit (first launch)	0.98	To earth orbit (second launch)	0.95
Ascend to lunar orbit	0.98	Rendezvous in earth orbit	0.95
		Rendezvous in lunar orbit	0.95
		Descend to lunar surface from orbit	0.95
		Direct earth arrival	0.95
Lower than reference risk		**Much higher than reference risk**	
Departure from earth orbit	0.99	Direct ascend from earth	0.9
Lunar orbit entry	0.99	Direct descend to lunar surface	0.9
		Ascend from lunar surface	0.9

Figure 6-5 Varying levels of mission risk

Instead of having engineers manually code up a unique total mass calculation routine for each mission mode, the ***Metric-Calc*** language automatically constructs a computable expression for each of the mission modes. It utilizes the locally defined transformation rules to incrementally modify the algebraic content of various instances of rocket equations. This symbolic manipulation mechanism eliminates the need to perform backward calculation. By assigning local variables and isolating them from the global context, dV_p and Isp_p can be evaluated within their local context or bound to a global variable at a later time during the enumeration process. As long as all the numerical values and calculation routines are defined, a numerical value would be calculated and returned. Otherwise, a structurally equivalent computable expression would be produced and the corresponding string representation would be returned. Using a graph-based structure to localize the transformation rules in ***Processes***, the expression construction process

simply follows the execution paths of ***Mission-Enum***. This approach is vastly different from propagating changes due to local variation. Most of these change propagation techniques only update dependent values in numeric terms [126]. ***Mission-Enum***, in contrast, can use the changed information to enumerate a new set of computable models, each representing a new mission mode. At the early stage of architecting, creating a set of computable models can be more revealing than just observing some numerical value change. In any case, OPN allows architects to dynamically construct new computable models and expressions. And these models and expressions can be evaluated into numeric values as sufficient knowledge becomes available.

6.1.4.4 Visualizing calculation results

Figure 6-6 shows the metric calculation results of two key decision metrics, weight of the total vehicle at launch time, and probability of mission success. Based on this two-dimensional data plot, decision makers can apply Pareto Front analysis to visually reason about their tradeoff decisions. For example, if one prefers lowest amount of risk, one might choose “EO+LO”, the mission mode that uses one vehicle that travels through both Earth Orbit and Lunar Orbit with no rendezvous operation. In contrast, one might choose to use “EOR+LOR” as an alternative mission mode, which requires the least amount of total vehicle weight for each rocket at launch time. The mission mode chosen in the Apollo Program was the “LOR+EO” alternative. As shown in Figure 6-6, the “LOR+EO” mission mode is also located on the Pareto Front. Via visual inspection, the “LOR+EO” alternative is significantly lighter than the “EO+LO” mission mode, while having a higher probability of success than all of the other alternatives.

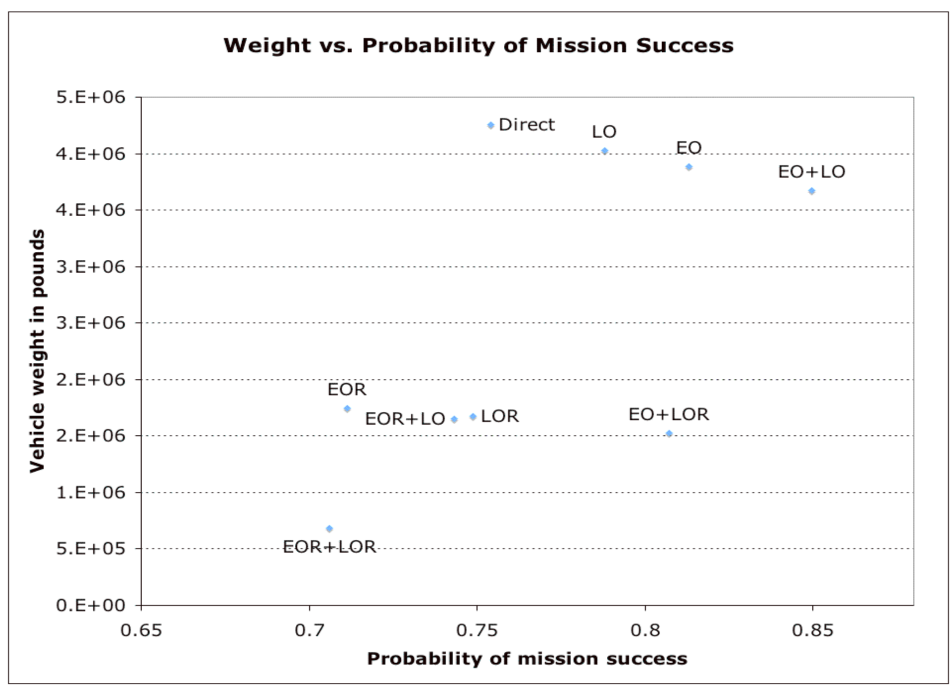

Figure 6-6 Visualizing a Two-Dimensional Metric Space

Evidently, other performance metrics can be calculated using the same mechanism. The language ***Metric-Calc*** may calculate many performance metrics as long as the computational knowledge and resources become available.

6.2 NASA's Space Exploration Initiative (current)

We also applied this modeling approach to NASA's Space Exploration Initiative [127]. The project is called: "Concept Evaluation and Refinement (CER) Project". The goal was to understand the space of possible mission architecture alternatives for space transportation vehicles that carry humans from Earth to the Moon or Mars. We will describe how OPN is used in this project below.

6.2.1 OPN as a mission-mode generator

The project started in September 2004, and within the course of three months we have constructed an executable model to enumerate the mission possibilities from Earth to Moon and Mars. We also identified many user interface improvement issues that were resolved in time to deliver useful results. In this project, we found that ***Mission-Space*** and ***Mission-Enum*** can be directly applied to serve the analytical tasks of this new NASA program.

The architecture analysis project includes a team of four people. Another twenty people developed Vehicle Models, ΔV Tool, Metric calculation, and LV Constraints Models. The project's workflow is illustrated in Figure 6-7.

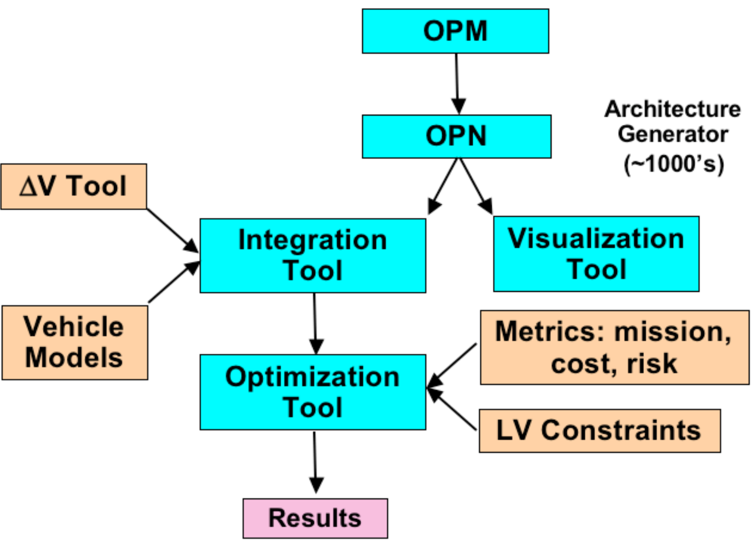

Figure 6-7 Workflow of the current NASA architecture study (Courtesy of Simmons)

One person was in charge of using OPN to create all the architectural alternatives. Due to memory and time considerations, various versions of ***Mission-Space*** (as described in Section 6.1.2) are created to generate up to 1000 mission architectures for each run. These models are very similar to the ***Mission-Enum*** model described in Section 6.1.3. OPN as a domain-neutral

modeling language allows other team members to cross-examine the assumptions made in different versions of OPN models. Once a reasonable ***Mission-Enum*** model is constructed, it generates all feasible architectural models.

6.2.2 OPN integrated with other software tools

The integration tool and optimization tool depicted in Figure 6-7 take the results produced by OPN in textual data format and feed them into different performance metrics calculation routines created by the other twenty collaborators. Figure 6-8 shows a diagram that represents a "family" of missions that are defined by one entry in OPN generated mission architecture. Each of the circles on this diagram represents a mission that fits a feasible Earth to Mars transportation architecture.

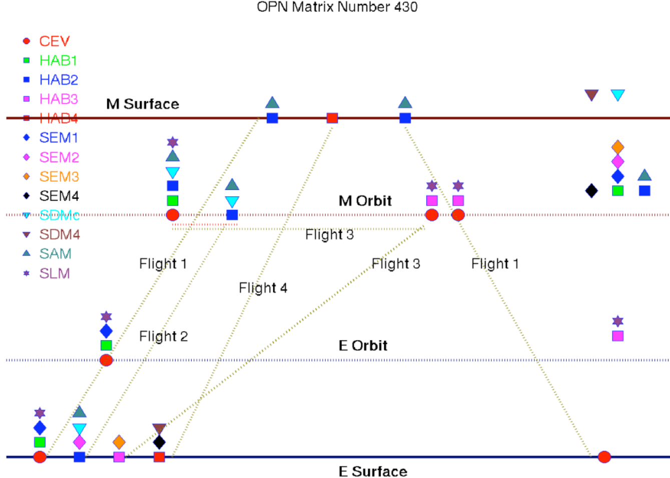

Figure 6-8 An architecture visualization tool driven by OPN's output (Courtesy of Simmons)

At the time of this study, the concept of ***Metric-Calc*** has not been fully demonstrated to the team working on the NASA project. Therefore, the team proceeded to create a separate set of tools that take ***Mission-Enum's*** generated mission architecture data to feed into a different tool. From

a software integration viewpoint, it is possible to incorporate the calculation of cost, risks, and ΔV calculation depicted in Figure 6-7 within OPN. The calculation example demonstrated in Section 6.6 of this thesis validates the feasibility.

6.2.3 Observation on OPN's usability

Three key findings resulted from this new NASA program:

1. *We found that OPN is capable of expressing most of the mission variation requirements without adding new language constructs.*
2. *We also found that by organizing missions into segments of a graph provides an intuitive way to manage complexity and communicate design ideas.*
3. *We found that it is very simple to refine the OPN model of space transportation as new knowledge becomes available. Incorporating new knowledge simply involves the adding, removing or changing of localized properties in **Objects**, **Process**, or **Pre/Post Conditions**.*

6.3 Enhanced Ground Testing Pod

Architects often need to make architectural decisions based on incomplete information. This section uses the Enhance Ground Testing Pod (EGT-Pod) as an example to illustrate how OPN supports the architectural reasoning process under incomplete information.

6.3.1 What is an EGT-Pod

EGT-Pod is an Inertia Measurement Unit (IMU) testing system under development at Draper Laboratory. An IMU is a sensor system that detects the position and altitude of a missile by measuring the accelerations and rotations applied to the missile's inertial frame. The IMU to be tested is a part of Trident missile's Mk6 guidance system. The objective is to perform non-destructive tests of IMUs by operating IMUs inside the Pod while the Pod is carried aloft by an F-15E Strike Eagle aircraft (see Figure 6-9).

Figure 6-9 The EGT-Pod and its carrying vehicle (Courtesy of Chris Anderson)

Trident missiles are designed to launch from nuclear submarines, operated by the Navy. Under certain contractual agreement with the Department of Defense, Draper Lab must design EGT-Pod based on aircraft operated by the Air Force such as the F-15E mentioned earlier. This design requirement creates additional logistic concerns. It raises the issues of coordinating equipment and personnel availability across two very large organizations. These logistic concerns also have impact on the architectural reasoning process. The goal is to incorporate these logistic concerns into the architectural reasoning process.

6.3.2 Alternative architectures of EGT-Pod

Two architectural alternatives are proposed. As shown in Figure 6-10, the Pod can be either mounted under the belly or under the wings of the aircraft. Different mounting configurations present different consequences in terms of aerodynamic performance. The belly mount option allows the aircraft to better accelerate during the test, therefore generating higher quality test data. In contrast, mounting under the wing allows each flight to test two Pods, therefore requiring fewer flights to complete the test operation. A shorter completion time for the test usually reduces the cost of flight operations and makes it easier to fit the schedule of available pilots and equipment.

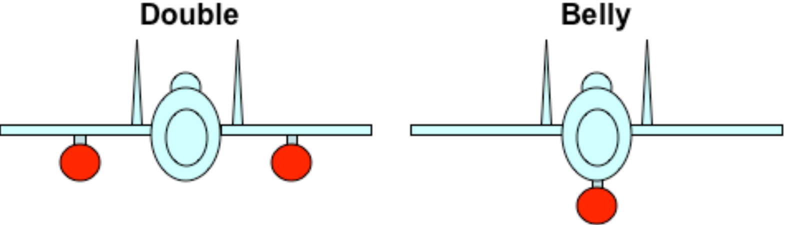

Figure 6-10 Two competing architectural alternatives

Choosing between "belly mount" and "double mount" is a challenge because precise relationships among customer preference, average test completion time, and data quality cannot be formulated ahead of time. However, this decision must be made during the early stage of the system development because changing it at later stages would imply significant design rework and incur changes on the logistic plan for the proposed test operations. These consequences motivate the architects of EGT-Pod to make a decision as early as possible. It also characterizes the fact that architects must reason through the consequences of decisions with incomplete information.

6.3.3 Infer global consequences from local knowledge

Given incomplete information, architects must present a strategy to acquire sufficient information to make an adequate decision. The decision would have been obvious, if the customer specifies that throughput (measured by the time required to complete a series of test) is always more important than data quality. The architect can simply choose the "double mount" configuration and proceed with the engineering development effort. However, most customers would usually demand the highest possible throughput and the best possible data quality. The lack of explicitly stated customer preference is often a challenge for architectural reasoning. However, by structuring incomplete information in a network structure, additional insight could emerge.

As shown in Figure 6-11, the Inference Result table shows a base scenario for the analysis. The "Customer Preference" ***Object*** is assigned with equal preference between "throughput" and "data quality". These two possible states of "Customer Preference" ***Object*** are

both assigned with fifty percent marginal probability. One can apply Bayes rule to calculate the marginal probability of preference order and expected value of other related variables. The calculation is based on the Bayes inference rule:

$$P(A, B) = P(A|B)\,P(B)$$

where

P(A,B) represents the joint probability of ***A*** and ***B***,

P(A|B) represents the conditional probability of event ***A*** given ***B***.

P(B) represent the probability of event ***B***.

By multiplying the marginal probability distribution of "Customer Preference" ***Object*** with the conditional probability table embedded in "Architecture Selecting" ***Process*** yields the marginal probability values of "Mount Configuration" ***Object***. The inference result for all four ***Objects*** in Figure 6-11 is summarized in the "Inference Result" table.

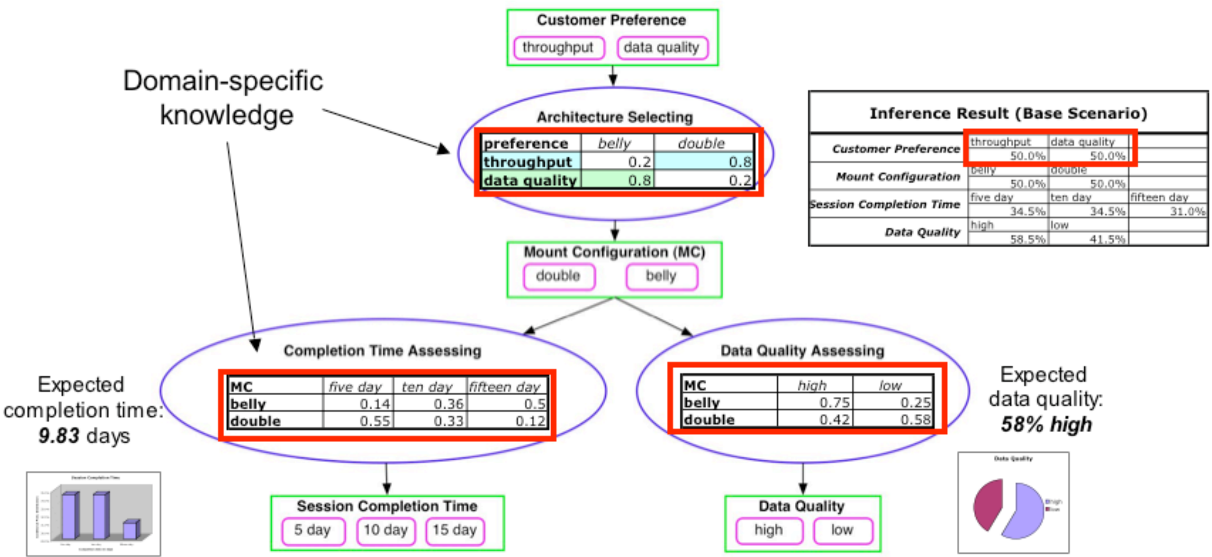

Figure 6-11 Probabilistically infer global effects given local knowledge

When the probability distribution function is associated to a quantitative measure the expected values can be estimated. Figure 6-11 shows that "Customer Preference" ***Object*** has an equal probability distribution. This indicates that the lack of customer preference between "throughout" and "data quality" makes it impossible to choose between "double" and "belly" system configuration. However, by continuously applying the statistical inference calculation, Figure 6-11 shows that "Session Completion Time" is estimated at 9.83 days and "Data Quality" has a 58 percent chance of getting "high" quality results. These inferred probability values provide physically or contextually meaningful quantities that can be presented to the customer or other stakeholders. It provides a communication instrument between architects and their customers to negotiate the what-if scenarios in contextually meaningful terms.

One may argue that it is difficult to formulate domain-specific knowledge in accurate probabilistic measures. As Glen Shafer once said: "Probability is not really about numbers, it is about the structure of reasoning." Moreover, it is relatively easy to extend the structure of the network by adding new nodes. Architects and stakeholders can incrementally adjust the network to fulfill their evolving understanding and communication needs. The probabilistic inference mechanism can support the communication process by computing the marginal probability for all ***Objects***. When necessary, simple calculation rules can be added to compute each ***Object's*** corresponding expected values. In other words, an OPN model equipped probabilistic inference engine allows architects and stakeholders to negotiate architectural decisions in quantitative terms under uncertainty.

To support architectural negotiation between non-technical stakeholders, the marginal probability values for all ***Objects*** can be updated and displayed graphically and interactively. The user interface design is discussed in Section 5.4.4.

6.3.3.1 Bidirectional inference mechanisms

OPN allows architects to utilize statistical data to infer system states bi-directionally. The OPN model in Figure 6-11 functions as a Bayesian Belief Network (BBN). In a BBN, the arrows do not indicate the directions of successive event sequences; they only indicate the direction of inference. The Bayes rule provides a formula to reverse the direction of inference. For example, given joint probability P(A,B) and marginal probability P(A) or P(B), one can calculate the corresponding conditional probability values. The calculation formula is shown as follows:

$$P(A|B) = P(A,B) / P(B) \qquad P(B|A) = P(A,B) / P(A)$$

BBN is a general-purpose tool for reasoning about decision under uncertainty. It allows architects to compose domain-specific knowledge in a network, and then infer direct statistical relationships among any set of variables in the network. This probabilistic inference mechanism is often called “belief propagation”. The algorithm can be implemented using OPN’s token generation and scheduling execution model. Kschischang et al. [69] shows how to realize belief propagation based on token generation and scheduling. Section 5.4.4 discusses the software implementation of belief propagation in OPN.

This chapter shows that OPN can serve as a formal language to specify, generate, and construct computable models of either dynamical or static systems. It also provides symbolic, numeric, and

probabilistic computational features to support architectural decisions. The following chapter will discuss OPN's intended roles in the field of system architecting.

7 Discussion

This chapter presents the contribution of this thesis to the field of system architecting.

7.1 Key Contributions

The main contribution of this thesis is to present a domain-neutral executable meta-language that ***automates certain architectural model construction tasks***. The automated tasks help reshape architectural reasoning processes on three operational levels. First, architects may utilize features of the meta-language to formally specify the space of possible systems. Conventionally, architectural reasoning processes rely on system description languages such as E-R model, UML, or other declarative modeling languages to specify system requirements. These languages conventionally lead the modeling effort to describe specific instances of systems. In contrast, OPN as a domain-neutral meta-language provides a more flexible and abstract vocabulary that allows architects and stakeholders to broadly specify the space of possible systems in formal terms. Second, executable meta-language allows architects to enumerate and generate executable system models. Currently, architects often use generative modeling techniques such as Genetic Algorithms or manually created morphological matrices to sample some parameterized representation of system models. OPN's model generation and enumeration features enable architects to automatically generate and enumerate executable system models. Third, it allows certain metric calculation routines to be composed automatically. Constructing simulation models and relevant performance metric calculation routines is often a tedious and time-consuming manual process. OPN's layered semantic model provides programming features that

automate certain model construction tasks. In many cases, complex metric calculation routines can also be automatically composed.

7.2 OPN addressing the needs in system architecting

In Chapter 1, this thesis argues that architectural decisions are derived from interactions among architectural decision-makers bounded by their respective knowledge and resource constraints. Wegner's Interaction Machine [24] and Gelernter's "Mirror Worlds" [128] concepts provided the inspiration to model complex systems in terms of communication and computation. Simon's "Bounded Rationality" [129] argument also influenced our thinking to model decisions based on available computational resources. Rational architects have the following communication and computational needs:

1. *An explicit representation of the space of decision alternatives to communicate with relevant stakeholders.*
2. *An efficient procedure to generate alternative architectural instances for more detailed scenario analysis and investigation.*
3. *An effective method to compute or assess the performance metrics that adequately reflects the variations in the generated architectural alternatives.*

In Section 6.1.2, we used OPN as a declarative language. It served as a representational medium to explicitly specify the space of decision alternatives. In Section 6.1.3, OPN was used as an imperative language. The token generating and scheduling algorithm automates the architectural instance generating procedures. Since OPN allows users to specify rules that control the token generating and scheduling events, users can apply domain specific knowledge to make the

alternative generation procedures focus on relevant subsets of architectural alternatives. In Section 6.1.4, we used OPN as a simulation language. It utilizes the variable binding mechanism to incorporate legacy code and domain-specific calculation routines to perform metrics calculation tasks.

7.2.1 OPN implementation meets the requirements

The OPN executable meta-language is a software tool designed to support architectural reasoning. The list of requirements for this tool is derived from our observation delineated in Section 3.3, and repeated here:

1. *Formally represent and specify the space of architectural alternatives by reflecting the knowledge of system variability across multiple knowledge domains*
2. *Automatically generate, enumerate and encode all instances of architectural alternatives specified in the meta-language*
3. *Adaptively calculate metrics associated with each generated architectural instance to help architects and other stakeholders perform tradeoff analysis on all instances of architectural alternatives*

These requirements are met and demonstrated in the examples illustrated in Chapter 6. Other related implementation requirements are:

1. *Subsume various models of computation*
2. *Computationally generate individual instances of architectural models*
3. *Mechanically construct metric calculation routines for each architectural models*
4. *Support layered abstractions to better integrate different types of system knowledge*

5. *Provide a diagrammatic user interface to facilitate human-machine interactions*
6. *Allow for the tool to be deployed across standard computing platforms*
7. *Enable both individual and collaborative modeling and simulation tasks*

In Chapter 5, we presented the software engineering aspects of OPN and showed that OPN can support the language requirements. It provides a layered semantic model to subsume different computational models, and its meta-language feature can generate individual instances of computable models or "object languages". It can mechanically construct arithmetic expressions for metric calculation. It supports a hierarchical data structure and layered semantic model to represent knowledge at different abstraction levels. It also provides a diagrammatic user interface that enables users to visualize the data content, structure, and algorithmic dynamics.

It also showed that OPN can be implemented and deployed using popular software implementation tools and it runs on machines that supports Java's J2EE standard. Its threading features and XML-based language model allows users to share models. OPN also provides a standard output mechanism to allow different tools to share simulation results.

7.2.2 Meta-language and qualitative methods

This thesis treats all aspects of design as language manipulation or language translation tasks. It aims to support the declarative, imperative, and simulation aspects of knowledge representation tasks using one flexible and executable language. In Section 2.1.2, we mentioned that existing system design methods such as Language Processing (LP), Design Structure Matrix (DSM), and Qualify Function Deployment (QFD) are useful in concept exploration and high-level system decomposition analysis. However, due to the qualitative nature of these methods, they usually

only provide directional guidance to a complex project. These methods can be useful in declaring a general direction of a design space, however, the semantics of these methods lack rigor to specify the dynamic properties of a complex interactive system. For example, DSM uses a binary matrix to represent the information flows or structural dependencies between subsystems. These binary matrices are often insufficient to encode the detailed information content embedded in subsystem relationships. Therefore, two systems with identical binary matrix markings may not have the same structural and behavioral qualities. Similar to DSM, QFD and Language Processing methods work for high-level system design. QFD provides a declarative vocabulary for stakeholders to establish a preference rank order of certain declared qualitative functions of a system. The preference order only provides some guidance on which function is more important than the others. It doesn't provide additional structural or behavioral description to inform further design activities. Language Processing Method is also useful in guiding the thought process of system designers. However, it is designed as a tool for visualizing the thought process, not as a tool for detailed technical analysis.

In contrast, an executable meta-language allows users to incrementally add additional structural and behavioral content as more detailed knowledge becomes available. In our implementation, OPN as a diagrammatic tool can also serve as an electronic blackboard for these qualitative analysis methods such as DSM, QFD and the Language Processing Method. As shown in Section 5.3 and 5.4, the relational dependencies between ***Objects*** and ***Processes*** can be visualized as a matrix. In other words, an executable meta-language can add user interface components to support DSM analysis as a way to communicate the structure of the system with stakeholders. Similar approaches can be applied to QFD analysis. In terms of the Language Processing Method,

OPN can serve as an electronic blackboard to support the process of extracting meaningful words and concepts in interactive brainstorm sessions. In other words, OPN as a graphical meta-language is designed to support the information processing needs in qualitative analysis methods.

7.2.3 Meta-language and quantitative methods

Quantitative design theories such as Axiomatic Design, and other quantitative methods based on Information Theory, Network Theory or Game Theory can all benefit from adopting a formal and executable language. These quantitative methods usually involve significant amount of mechanical calculation tasks. For example, Axiomatic Design requires the rearrangement of the matrix as well as the calculation of information content for each of the alternative designs. These calculation tasks can be automated based on the algorithm specification provided with the theory. At the same time, Information Theory, Network Theory, and Game Theory often involve sophisticated probability or payoff function calculations. A graphical and executable language such as OPN can incorporate these calculation routines and support these design theories on an operational level. In Section 6.1.4, we showed that OPN calculated mission success probability as a Markov Network [66]. The structure of a tradeoff decision can also be formulated as a "graphical game" [42], which is a variation of a graphical probabilistic model. In Section 2.2.4 and Section 4.8, we discussed how probabilistic graphical models can be incorporated into the meta-language execution engine. In general, when working on quantitative design evaluation tasks, an executable meta-language serves the function of a simulation language to assess the

global consequences of system interactions. In Chapter 5 and 6, we showed that OPN is designed to support both qualitative and quantitative design analysis methods.

7.3 OPN as an executable meta-language

OPN is a graphical executable meta-language designed for system architects. The design of OPN is influenced by ML and Lisp. For example, the notion of having a meta-operand and meta-operator in OPN is a direct descendant of Lisp. However, ML and Lisp are textual programming languages designed for mathematicians and computer scientists; they are not suitable for communicating high-level system architectural ideas with non-technical stakeholders. We were also influenced by Category Theory, which is a graphical meta-language for mathematics, but not an executable language. We found Category Theory's concept of manipulating mathematical functions and entire classes of mathematical objects as the operands of a reasoning process can be useful to system architects. It provides a formal framework to reason about the relationships between functions and forms in a complex system. Certain transformation rules in Category Theory such as function composition, and natural transformation can be made executable. These executable features are implemented in OPN.

7.3.1 OPN and pattern languages

Pattern languages are mostly designed to be declarative languages. They help architects to decompose a larger problem into smaller chunks of well-understood design patterns. However, most pattern languages lack the imperative language feature to specify how to compose various patterns into a specific instance of design. Therefore, pattern languages can be used as the basic

vocabulary for the declarative aspect of OPN. Then, OPN's imperative language features can be utilized to automatically generate possible instances of architectures. OPN provides a generic imperative model to combine and evolve patterns in pattern languages. OPN is a generative model; it only requires users to specify the space of possible alternatives. The token generation and scheduling algorithm in OPN will try to enumerate all possible instances of architectures. Different instances of architectures are computable models by themselves and can be further refined and compared to derive more variations as users interact with them. In other words, by combining pattern languages with generative modeling features, OPN can be used by pattern language practitioners to explore system design in an automated fashion.

7.3.2 OPN as a system description language

System description languages such as E-R diagram, UML and OPM laid the foundational work in modeling complex socio-technical systems. They demonstrated that graphical models could be practically deployed in system development, software modeling and, to certain extent, automatic code generation for database systems or real-time control systems. They also have obvious limitations. As mentioned in Section 2.2.2, E-R diagram primarily focuses on the static relationships of a system. It doesn't provide a model of computation for specifying the evolving nature of the system.

7.3.2.1 OPN vs. UML

UML is a highly complex language that intends to cover most of the needs in complex system representation. Unfortunately, the complexity of the language itself has become a serious hindrance to perform model integration. For simple systems, UML models can be more complex

than writing and debugging source code of some simple programming languages. UML contains a large number of sub languages. Integration between languages is often not supported formally. To provide formal support in language integration, UML manages language definitions by declaring a four-layered meta-model architecture. This meta-model architecture is static in nature, and it requires a centralized revision committee to make modification on the meta-model level. To address the language bloat problem in UML, OPN allows users to work with ***one*** executable meta-language to support both language definition and model execution tasks. An ideal system description and simulation language should avoid introducing unnecessary notations and model syntax to complicate the architectural reasoning tasks. Instead of forcing users to perform model integration tasks and learning new language standards, OPN helps users to directly focus on the description and simulation of domain specific problems.

7.3.2.2 OPN vs. OPM

OPM is a system description language designed to address the needs in simultaneously representing a system's structural and behavioral properties. It avoids the "language bloat" problem that often stifles the development of language standards such as UML. However, OPM's Object-Process Diagram (OPD) allows users to specify many types of graphical relationships between ***Objects*** and ***Processes*** that can be overwhelming to novice users.

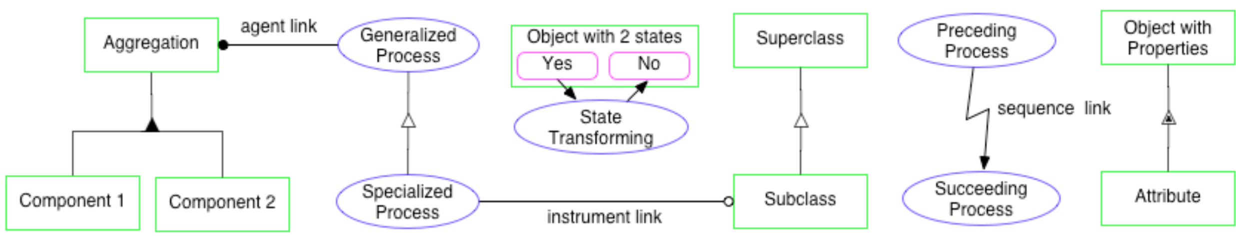

Figure 7-1 OPM's different link types

To avoid "notation bloat" in the diagrammatic language, OPN's graphical notation strictly follows the bi-partite graph formalism. It only allows direct connections between ***Object-Process*** pairs. Only two types of relationships are allowed in OPN; ***Pre-Condition*** specifies the relationship from ***Object*** to ***Process***. ***Post-Condition*** specifies the relationship from ***Process*** to ***Object***. All relationship types in OPD are emulated through customizable data structures associated with the respective ***Object***, ***Process*** and ***Pre/Post Conditions***. This bi-partite graphical formalism provides a consistent execution language schema that associates computable functions with ***Processes*** and stores computational results into ***Objects***.

Language features in OPM that are not absolutely needed by the meta-language are not incorporated into OPN. For example, the "Formal English" generating feature, also known as Object-Process Language (OPL) is not incorporated into OPN because it can be added externally without affecting the structural and behavioral properties of the system of interest. The structural link, specialization link, enabler link, and agent link are all removed because a pair of "***Pre/Post Relationships***" and a "***Process***" can be customized to computationally emulate their semantic meaning. OPN is intentionally designed as a minimalist meta-language; it tries to avoid semantic definitions that can be composed from more basic linguistic constructs. The goal of this minimalist design approach is to make the language definition small, so it would require less implementation and debugging effort to create the execution environment.

7.3.3 OPN and generative modeling techniques

OPN's simple language structure uses one meta-operand, ***Thing***, and one meta-operator ***Eval***. This meta-operator and meta-operand pair of OPN provides a consistent programming interface

that is particularly suitable for generative modeling. Its inputs, outputs, and execution algorithms are all modeled as ***Things***. The processes of generating and modifying ***Things*** are modeled by various customizable versions of ***Eval***. The concept of meta-operand and meta-operator pair is derived from functional programming language such as Lisp, Mathematica, and ML. These functional programming languages have served well in performing generative modeling such as writing code for self-modifying genetic algorithms [62] and performing computational experiments for interacting cellular automata [25]. However, these programming languages require significant programming skills, and we found it possible to graphically represent some of the abstract functional programming concepts, such as recursion and symbolic variable replacement using a graphical programming language. Allowing users to visualize the model generating process enables architects and non-programmers to perform model generation tasks without the need to construct models manually. OPN's user interface design also enables architects to graphically construct and debug model generation "programs". The user interface also helps system architects and stakeholders to visually explore solution alternatives by inspecting the generated structures and computational results interactively.

Moreover, OPN allows GA experts and functional programming experts to formulate their problems in a graphical programming environment. The tokens in OPN can be treated as the mutating genes or the dynamically evolving computable expressions. It can be used as a platform to create the structure of these generative algorithms. When necessary, these generative algorithms can be first developed in OPN and then manually transcribed into source code of some high performance programming language to perform more extensive computational experiments.

7.3.4 OPN as a simulation language

OPN as a general-purpose programming language can be customized to incorporate various algorithms and models of computation to support architects' reasoning tasks. The execution model of OPN is based on a three-layered semantic interpretation engine. The top layer mimics Petri Net's token scheduling rules. The token processing layer using inference rules to perform token transformation tasks. The token processing layer invokes individual rules or functions specified by stakeholders with process-level domain expertise. Simple algebraic equations or arithmetic expressions can be specified on the process level to perform calculation tasks. The variable binding layer allows users to access customized algorithms or perform communication with other computational services on the network. These customized algorithms and third party simulation code provide an additional layer of functionality to allow users to perform more sophisticated simulation tasks. To perform probabilistic reasoning, OPN may incorporate the message propagation algorithm [66] to perform bi-directional inference. For System Dynamic simulation, OPN's token generation and scheduling mechanism and process transformation routines also allow users to perform numerical integration tasks and compose arithmetic expressions over multiple time intervals. For discrete event simulation, OPN can use the standard Petri Net token scheduling algorithm specified in Section 4.6 to perform discrete event simulation. In other words, OPN can serve as a hybrid simulation environment that integrates the functionalities of probabilistic, numeric, symbolic, and discrete event simulation engines.

7.3.4.1 OPN vs. Petri Net

OPN's graph formalism for the event generation and scheduling aspect is particularly similar to Petri Net. However, OPN subsumes Petri Net and other graphical formalisms in the following ways:

1. ***OPN is a meta-Petri Net***: During model execution time, OPN records the execution sequences of all tokens within the tokens themselves as computationally generated OPN models. Unlike Petri Net, each token firing event is simply a computational abstraction of some simulated external activities. OPN's tokens not only record the state information during runtime, they also add the Objects, Processes and Relationships that created them into a locally stored OPN model. In other words, each OPN token processing event is a model construction activity. In the Petri Net literature, this meta-modeling extension is sometimes referred to as Higher Order Petri Net [82, 83].

2. ***Objects vs. Places***: OPN organizes closely related variables into Objects; places in Petri Net are represented as Objects in OPN that capture state information and complex data structures at the Object level. Instead of defining "Places" as passive storage of tokens, the notion of Object provides additional graphical information on each instance of Object to visualize the state of a system during model execution time [26]. This approach is different from Colored-Petri Nets since "color" in "Colored Petri Net" refers to the value of the token, not the "place". Objects with visible local information yield a more convenient visual formalism for continuous and probabilistic systems.

3. ***Processes vs. Transitions***: ***Process*** in OPN replaces Transition in Petri Nets. A "transition" in Petri Net represents an action to occur in time. A ***Process*** in OPN refers to a mathematical function or relationship in general. It is not bound to an action in the time domain. When OPN is used as a model enumeration engine, the notion of "time" or "event sequence" can be conceptually ignored, the enumeration algorithm is simply listing combinatorial structures that may or may not contain the notion of time. When a ***Process*** is included in the generated model, it denotes a possible state-space mapping between its neighboring ***Objects***. This "mapping" is not considered to be an "action", but a declared function or a binding constraint between the state-spaces of its neighboring ***Objects***.

7.4 Future Development

This section presents the future research opportunities based on the meta-language approach described in this thesis. The future development activities are divided into three broad categories: theory, tool, and application development.

7.4.1 Theory development

Two areas of theory development are of interest in the context of complex system architecting. The first one is related to statistical network theory. The second one is related to the mathematical properties of complex system models.

7.4.1.1 Statistical network theories and system architecting

The concept of analyzing systems as a complex network is gaining popularity in both scientific and engineering research communities. Typical approaches are based on analyzing the statistical properties of these networks. Researchers have attempted to draw conclusions from certain generalized statistical qualities of complex networks in terms of small world effects [118, 130], scale-free networks [131-133], and power laws [134]. The explicit use of these methods for designing complex systems have not been wide spread, partially because these aggregate properties hide the detail information about context-specific design concerns. However, these statistical properties provide some insight into very large-scale networked systems. This is an area of theoretical development that is outside the scope of this thesis.

7.4.1.2 The mathematical properties of complex system models

One must also note the limitation of statistical network theories when applying them to reason about complex system models. A statistically based network analysis ignores context-specific information because it treats all relationships between different nodes uniformly. A relationship that connects two nodes in a network may contain useful and unique information that can change the behavior of the entire network. The analytical method must be able to identify and extract this information whenever necessary. A statistical approach to networks would not be able to accommodate this fine-grained approach. Most statistical procedures simply count the connectivity between nodes or assign some numeric measure to each node or arc. They completely ignore the intricate data structure that might be associated with individual nodes and relationships.

Modeling a network as an instance of language avoids the problem of overlooking the details. However, that implies two additional issues that must be addressed.

1. *The data structure required to capture the detailed information*
2. *The required amount of storage space and time to manage this added information*

The first problem can be addressed by using a recursive data structure, such as the one we proposed in this thesis to capture all levels of details.

The second problem requires more attention. First, storage space required to capture this additional information can be compressed by properly classifying the types of nodes and links. This concept is a well-known technique in Object-Oriented programming [135]. Gabriel [44] and Whitmire [136] have stated their views on how significant compression in terms of the size of models and the effort required to build the model can be addressed and measured using Object-Oriented modeling techniques. Applying techniques based on Type-Calculus [137] provides a method to manipulate information on classes or types of object, and also provides a mechanism to compress the processing time required to analyze the data. Computational inference techniques can be applied to manipulate data based on instances of class data entries [120], not the instances of data entries on nodes or links. To enable these types of analysis requires a modeling language that can directly operate on typing information about its internal data structures. Many modern programming languages provide these features. The issue is that using these features requires significant programming skills. An alternative is to reveal these features in an intuitive manner, so that non-programmers can utilize these features to manipulate data structures. This is achievable by OPN, but not demonstrated in this thesis. A detailed case study showing how network-based language can enable a wide range of people to perform

computationally assisted reasoning on the types, not just the instances of data entries, may have the potential to be a highly fruitful research direction.

7.4.2 Tool development

Without adequate instrumentation, theories can hardly be verified and conveyed in convincing manner. Therefore, ongoing development on the language manipulation tools is a critical area of research as well. The following features in a meta-language modeling environment are highly desirable:

1. *Estimate the memory and time requirements before enumerating alternatives*
2. *Improve speed and memory efficiency on model/object language generation routines*
3. *Provide access to third party computational resources*

All the suggested research areas are challenging research topics on both technical and theoretical levels. The concept of sizing a problem space in common space/time terms provides an upper bound to determine when and how to decompose the problem into a meaningful size for thorough architectural alternative study. The second problem is a practical issue that requires innovative techniques in combinatorial algorithm development. Designing efficient algorithms to generate all the sub-structures of a model helps us understand the nature of the model [59]. The third area is an engineering problem. Integrating legacy resources is always a necessary and highly profitable area of system development because it provides continuity in the operational environment, and reduces duplicate effort during tool development.

7.4.3 Application Development

OPN is a brand new language. It needs an active user community to provide design feedback, develop use cases for different application domains, and create supporting libraries to enrich its functionalities. To increase the size of user community, the following developmental activities are suggested:

1. *Develop self-contained tutorial that leads non-programmer to use OPN as a reasoning tool*
2. *Quantify in economical terms the benefits of using OPN*
3. *Create a secure data service on the Internet for OPN model/knowledge sharing*

This chapter presented the contribution of this research and compared the solutions proposed, implemented and demonstrated in the thesis with already existing solutions. The following chapter presents our conclusion.

8 Conclusion

This thesis presented a domain-neutral executable meta-language, Object-Process Network (OPN), which automates certain mechanical communication and computational tasks in architectural reasoning. The introductory remarks of this thesis first provided an operational definition of system architecting and complex socio-technical systems. Chapter 1 also articulated the rationale behind modeling the architectural reasoning process in terms of computation and computational tasks. It also stated the research opportunities in creating a domain-neutral software instrument for system architects based on the theory of ***communication and computation***.

Chapter 2 of this thesis presented the prior art related to the representation and analysis of complex socio-technical systems. It showed that current theories and methods of system architecting are often designed to fit specific knowledge domains [138, 10, 56, 139], limiting their expressiveness to incorporate knowledge that originates from different domains. It also showed that general-purpose modeling languages usually contain a large set of vocabulary and syntactic rules, requiring a significant learning effort. Other language-based architectural reasoning techniques such as pattern languages [2, 43, 13, 44, 45] only capture design heuristics that usually lack formal models of computation, thereby providing limited reasoning power when applied to complex and ambiguous architecting scenarios. Research work that focuses on unifying formal models of computation [137, 140, 15, 75, 141-143, 78] presents rigorous and domain-neutral architectural reasoning techniques. However, they are often illustrated in pure mathematical abstraction, making them less accessible to architects without extensive training in mathematical reasoning.

Chapter 3 presented the arguments showing that architects need a domain-neutral executable meta-language to improve existing architectural reasoning processes. It consolidates principles and knowledge presented in Chapter 2 into a requirement list for building an operational instrument for architectural reasoning. The key design ideas of the meta-language are:

1. *Specify a simple and stable syntax to enable domain-neutral knowledge exchange. This language feature helps streamline many repetitive* ***communication*** *tasks.*
2. *Design a simple and stable execution semantic model that satisfies Turing Completeness. This language feature allows architects to conduct variable kinds of* ***computational*** *tasks in an integrated environment.*
3. *Organize the semantic model into a layered structure, so that different aspects of system complexity can be temporarily suppressed. This* ***layered semantic model*** *reduces users' cognitive burden when constructing, navigating, and manipulating system models.*
4. *Use network-like diagrams (graphs) to visualize system structure and behavior.* ***Graphs*** *are treated as the basic building blocks of the language.*
5. *Use* ***one meta-operand and one meta-operator*** *to build the language kernel. This provides structural and behavioral consistency across all systems modeled using this meta-language.*

Chapters 4 and 5 presented the formal syntax, execution semantics, and software implementation pragmatics of the meta-language, OPN. Chapter 4 focuses on the implementation neutral aspect of the language architecture. It was intentionally written to be independent of implementation concerns. The concept is that a meta-language should not be

dependent on its underlying implementation technologies. Chapter 5 focuses on the software engineering considerations and the design rationale of important user interface features. It shows the feasibility of implementing a domain-neutral, graphical, and executable meta-language using existing off-the-shelf technologies. Information in Chapters 4 and 5 only specifies the syntax and domain-independent (abstract) semantics of OPN. The domain-dependent (concrete) semantics of OPN is later specified in Chapter 6, showing that the abstract OPN can be applied to different application domains.

Chapter 6 presented three case studies to illustrate how an executable meta-language can be utilized in architectural reasoning processes. The result of these case studies indicates that many aspects of computational and communication needs in complex system architectural reasoning can be satisfied by OPN. It shows that OPN can either replace or co-exist with many existing modeling languages and computational tools. More importantly, it helps shift the mental model of system architects in three significant ways:

1. *OPN's abstract vocabulary enables architects to flexibly* ***specify the space of architectural alternatives****, instead of trying to specify instances of architectures pre-maturely.*
2. *OPN enables architects to systematically* ***generate and enumerate executable system models****. In the past, models of alternative architectures are often manually crafted and randomly sampled.*
3. *OPN's symbolic programming features help system modelers to compose certain* ***complex metric calculation routines automatically****. This feature allows architects to better assess performance metrics of certain classes of complex models without investing a significant amount of modeling labor.*

Chapter 7 provided a detailed account of the main contribution of this thesis. It compared OPN to other modeling techniques and simulation languages. It showed that an abstract meta-language kernel embodies many desirable features of different system design methods and modeling languages. It argues that many repetitive reasoning tasks can be modeled and automated using an executable meta-language. It also showed that OPN as an executable meta-language satisfies many functional requirements of different modeling languages; therefore, it can either replace or emulate other modeling languages when appropriate.

In summary, this thesis argues that the science and technologies of language manipulation can be systematically utilized to improve the practice of system architecting. In this thesis, we showed that many architecting processes are composed of three types of modeling tasks; each of them corresponds to a type of language. They are:

1. *Declarative language: Specify the space of architecture alternatives*
2. *Imperative language: Enumerate architecture instances*
3. *Simulation language: Calculate the performance metrics of a given architecture instance*

This thesis also showed that OPN, a graphical meta-language language, can be implemented and deployed to carry out modeling tasks in practical applications. As a modeling language, OPN influences architects' reasoning processes in three systematic ways:

1. *OPN shifts the modeling focus from specifying instances of architectures to* ***specifying the space of possibilities***
2. *OPN offers many* ***model generation and enumeration*** *features that permit architects to programmatically explore the solution space*

3. *OPN's meta-language features makes it possible to* ***automatically compose performance metrics calculation routines*** *that are often tedious and difficult to do in other modeling languages*

The objective of this thesis is to formulate a domain-independent reasoning technique in terms of communication and computation. This thesis substantiates this claim by showing that OPN, a domain-independent modeling language, can be employed by architects to communicate design intent, construct simulation models, and compute performance metrics. The aim of this thesis is to provide a solution for the automation of the architectural reasoning tasks. Based on the analysis of the needs and the case study results, OPN satisfies the automation needs documented in the thesis, and therefore qualifies to be a solution for architectural reasoning tasks.

9 Bibliography

[1] C. Alexander, *The timeless way of building*. New York: Oxford University Press, 1979.
[2] C. Alexander, *The nature of order: an essay on the art of building and the nature of the universe*. Berkeley, Calif.: Center for Environmental Structure, 2002.
[3] S. Mac Lane, *Mathematics, form and function*. New York: Springer-Verlag, 1986.
[4] T. Berners-Lee, "Web Architecture from 50,000 feet," vol. Dec. 2004. Cambridge. MA: World Wide Web Consortium, 1998.
[5] R. M. Henderson and K. B. Clark, "Architectural Innovation - the Reconfiguration of Existing Product Technologies and the Failure of Established Firms," *Administrative Science Quarterly*, vol. 35, pp. 9-30, 1990.
[6] D. E. Ingber, "The architecture of Life," *Scientific American*, pp. 48-57, 1998.
[7] L. Lessig, *Code: and other laws of cyberspace*. [New York, N.Y.]: Basic Books, 1999.
[8] J. M. Utterback, *Mastering the dynamics of innovation: how companies can seize opportunities in the face of technological change*. Boston, Mass.: Harvard Business School Press, 1994.
[9] P. Carlile, "Transferring, Translating, and Transforming: An Integrative Framework for Managing Knowledge Across Boundaries," *Organization Science*, vol. 15, pp. 555-568, 2004.
[10] R. M. Murray, "Panel on Future Directions in Control, Dynamics, and Systems," California Institute of Technology, 2002.
[11] A. N. Whitehead, *An introduction to mathematics*. New York: H. Holt and company, 1911.
[12] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and interpretation of computer programs*, 2nd ed. Cambridge, Mass.: MIT Press, 1996.
[13] C. Y. Baldwin and K. B. Clark, *Design rules*. Cambridge, Mass.: MIT Press, 2000.
[14] Y. Ben-Haim, *Information-gap decision theory: decisions under severe uncertainty*. San Diego, Calif.: Academic Press, 2001.
[15] A. W. Burks, *Chance, cause, reason; an inquiry into the nature of scientific evidence*. Chicago: University of Chicago Press, 1977.
[16] M. Resnick, *Turtles, termites, and traffic jams: explorations in massively parallel microworlds*. Cambridge, Mass.: MIT Press, 1994.
[17] W. R. Ashby, *An introduction to cybernetics*. London: Chapman & Hall, 1961.
[18] E. Borger, *Architecture design and validation methods*. Berlin; New York: Springer, 2000.
[19] B. Chandrasekaran, J. Glasgow, and N. H. Narayanan, *Diagrammatic reasoning: cognitive and computational perspectives*. Cambridge, Mass.: MIT Press, 1995.
[20] J. S. Dahmann, R. M. Fujimoto, and R. M. Weatherly, "The DoD High Level Archiutecture: An Update," presented at Winter Simulation Conference, 1998.
[21] J. H. Holland, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*, 1st MIT Press ed. Cambridge, Mass.: MIT Press, 1992.
[22] J. H. Holland, *Emergence: from chaos to order*. Reading, Mass.: Addison-Wesley, 1998.
[23] T. S. Kuhn, *The structure of scientific revolutions*, [2d, enl. ed. Chicago]: University of Chicago Press, 1970.
[24] P. Wegner, "Why interaction is more powerful than algorithms," *Communications of the ACM*, vol. 40, 1997.
[25] S. Wolfram, *A new kind of science*. Champaign, IL: Wolfram Media, 2002.
[26] D. Dori, *Object-Process Methodology: A Holistic Systems Paradigm*: Springer-Verlag, 2002.
[27] T. S. Kuhn, J. Conant, and J. Haugeland, *The road since structure: philosophical essays, 1970-1993, with an autobiographical interview*. Chicago: University of Chicago Press, 2000.
[28] J. Moses, "The Anatomy of Large Scale Systems," in *Engineering Systems Internal Symposium*: MIT ESD, 2002.
[29] A. Asperti and G. Longo, *Categories, types, and structures: an introduction to category theory for the working computer scientist*. Cambridge, Mass.: MIT Press, 1991.
[30] S. Mac Lane, *Categories for the working mathematician*, 2nd ed. New York: Springer, 1998.

[31] B. C. Pierce, *Basic category theory for computer scientists*. Cambridge, Mass.: MIT Press, 1991.
[32] S. Abramsky and A. Jung, "Domain theory," in *Handbook of Logic in Computer Science*, vol. III, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds.: Oxford University Press, 1994.
[33] D. Scott, "Data types as lattices," presented at International Summer Institute and Logic Colloquium, Kiel, 1975.
[34] N. Wiener, *Cybernetics: or, Control and communication in the animal and the machine*, 2d paperback ed. Cambridge, Mass.: M.I.T. Press, 1965.
[35] D. Clausing, *Total quality development: a step-by-step guide to world class concurrent engineering*. New York, NY: ASME Press, 1993.
[36] G. S. Altshuller, *Creativity as an exact science: the theory of the solution of inventive problems*. New York: Gordon and Breach Science Publishers, 1984.
[37] S. D. Shiba, A. Graham, D. Walden, T. H. Lee, R. Stata, and Center for Quality Management (Cambridge Mass.), *A new American TQM: four practical revolutions in management*. Cambridge, Mass.; Norwalk, Conn.: Productivity Press, 1993.
[38] S. D. Shiba, D. Walden, and S. D. Shiba, *Four practical revolutions in management: systems for creating unique organizational capability*. Cambridge, Mass.: Center for Quality of Management, 2001.
[39] N. P. Suh, *The principles of design*. New York: Oxford University Press, 1990.
[40] N. P. Suh, *Axiomatic design: advances and applications*. New York: Oxford University Press, 2001.
[41] H. W. Kuhn and S. Nasar, *The essential John Nash*. Princeton, N.J. Chichester: Princeton University Press, 2002.
[42] M. Kearns, M. Littman, and S. Singh, "Graphical models for game theory," *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pp. 253-260, 2001.
[43] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: towns, buildings, construction*. New York: Oxford University Press, 1977.
[44] R. P. Gabriel, *Patterns of software: tales from the software community*. New York: Oxford University Press, 1996.
[45] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Boston, Mass.: Addison-Wesley, 1995.
[46] OMG, "Unified Modeling Language (UML), version 1.5," vol. 1: Object Management Group, 2003.
[47] P. Rivett, "UML 2.0 Infrastructure Final Adopted Specifcation," vol. 1: Object Management Group, 2004.
[48] OMG, "Unified Modeling Language (UML), version 1.4," vol. 1: Object Management Group, 2001.
[49] B. Selic, "UML 2.0 Superstructure Final Adopted specification," vol. 1: Object Management Group, 2004.
[50] B. Selic, "UML 2.0: Exploiting Abstration and Automation," in *Software Development Times*: SD Times, 2004.
[51] D. Dori, "Why Significant Change in UML is Unlikely," *Communications of ACM*, pp. pp. 82-85, 2002.
[52] M. Peleg and D. Dori, "The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods," *IEEE Trans. Software Engineering*, vol. 26(8), pp. 742-759, 2000.
[53] N. R. Soderborg, E. F. Crawley, and D. Dori, "*System function and architecture: OPM-based definitions and operational templates*," *Communications of ACM*, vol. 46(10), pp. 67-72, 2003.
[54] D. Dori, I. Reinhartz-Berger, and A. Sturm, "Developing Complex Systems with Object-Process Methodology with OPCAT," presented at Conceptual Modeling - ER 2003, 2003.
[55] M. Plusch, "Water: simplified Web services and XML programming." Indianapolis, Ind.: Wiley Pub., 2003.
[56] A. J. M. Rocha and Massachusetts Institute of Technology. Dept. of Architecture., "Architecture theory, 1960-1980: emergence of a computational perspective," 2004, pp. 175 leaves.
[57] M. Leyton, *A generative theory of shape*. Berlin; New York: Springer, 2001.
[58] M. Cook, "Mathew Cook showing Rule 110 is Turing Complete," vol. Dec. 2004: Wikipedia, 1998.
[59] L. A. Goldberg, *Efficient algorithms for listing combinatorial structures*. Cambridge; New York: Cambridge University Press, 1993.
[60] J. H. Holland, *Induction: processes of inference, learning, and discovery*. Cambridge, Mass.: MIT Press, 1986.
[61] J. H. Holland, *Hidden order: how adaptation builds complexity*. Reading, Mass.: Addison-Wesley, 1995.
[62] K. Sims, "Evolving Virtual Creatures," *Computer Graphics*, pp. 15-22, 1994.

[63] I. L. Kim and O. d. Weck, "Variable Length Chromosome Genetic Algorithm for Progressive Refinement in Topology Optimization," *Structure and Multidisciplinary Optimization*, vol. 1, 2005.
[64] C. Jacob, *Illustrating evolutionary computation with Mathematica*. San Francisco: Morgan Kaufmann Pub., 2001.
[65] B. P. Zeigler, *Theory of modelling and simulation*. New York: Wiley, 1976.
[66] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. San Mateo, Calif.: Morgan Kaufmann Publishers, 1988.
[67] J. Pearl, "Belief Networks Revisited," *Artificial Intelligence*, vol. 59, pp. 49-56, 1993.
[68] J. Pearl, "Decision making under uncertainty," *Acm Computing Surveys*, vol. 28, pp. 89-92, 1996.
[69] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor Graphs and the Sum-Product Algorithm," *IEEE Transactions on Information Theory*, vol. 47, 2001.
[70] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Englewood Cliffs, N.J.: Prentice Hall, 1995.
[71] E. Charniak, "Bayesian Networks without Tears," in *AI magazine*, 1991.
[72] M. Jordan, Z. Ghaharamani, T. Jaakkola, and L. Saul, "An introduction to variational methods for graphical models.," in *Learning in Graphical Models.*, M. Jordan, Ed., 1998.
[73] R. Detcher, "Bucket elimination: a unifying framework for processing hard and soft constraints," *ACM Computing Surveys*, vol. 28, 1996.
[74] F. G. Cozman, "Generalizing Variable Elimination in Bayesian Networks," *Proceedings of the IBERAMIA/SBIA 2000 Workshops*, pp. 27-32, 2000.
[75] H. Ehrig, *Unifying Petri nets: advances in Petri nets*. Berlin London: Springer, 2001.
[76] Z. Ghahramani, " Learning Dynamic Bayesian Networks," in *Adaptive Processing of Sequences and Data Structures*, C. L. G. a. M. G. (eds.), Ed. Berlin: Springer-Verlag, 1998.
[77] R. E. Neapolitan, *Learning Bayesian networks*. Harlow: Prentice Hall, 2003.
[78] C. A. Petri, "Kommunikation mit Automaten," vol. Ph.D.: University of Bonn, 1962.
[79] J. L. Peterson, *Petri net theory and the modeling of systems*. Englewood Cliff, N.J.: Prentice-Hall, 1981.
[80] X. Li, W. Yu, and F. Lara-Rosano, "Dynamic Knowledge Inference and Learning under Adaptive Fuzzy Petri Net Framework," *IEEE Trans. on Systems, Man, and Cybernetics*, 2000.
[81] J. W. Janneck and R. Esser, "Higher-order Petri net modeling:techniques and applications," *23rd International Conference on the Application and Theory of Petri Nets*, 2002.
[82] C. A. Lakos, "Object Petri Net: Definition and Relationship to Coloured Petri Net," *Computer Science Department*, vol. TR94-3, 1994.
[83] C. A. Lakos, "From Coloured Petri Net to Object Petri Net," *Proceedings of the 15th International Conference on the Application and Theory of Petri Nets*, vol. 815, 1995.
[84] J. W. Forrester, *Industrial dynamics*. Waltham, MA: Pegasus Communications, 1999.
[85] J. Sterman, *Business dynamics: systems thinking and modeling for a complex world*. Boston: Irwin/McGraw-Hill, 2000.
[86] T. W. Malone, K. Crowston, and G. A. Herman, *Organizing business knowledge: the MIT process handbook*. Cambridge, Mass.: MIT Press, 2003.
[87] J. Patten, H. Ishii, J. Hines, and G. Pangaro, "Sensetable: a wireless object tracking platform for tangible user interfaces," *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2001.
[88] K. Nygaard and O.-J. Dahl, "The development of the SIMULA languages," *ACM SIGPLAN Notices, The first ACM SIGPLAN conference on History of programming languages*, vol. 13, 1978.
[89] T. W. Christopher, *Python programming patterns*. Upper Saddle River, NJ: Prentice Hall, 2002.
[90] T. Berners-Lee and M. Fischetti, *Weaving the Web: the original design and ultimate destiny of the World Wide Web by its inventor*, 1st ed. [San Francisco]: HarperSanFrancisco, 1999.
[91] T. Berners-Lee, "Semantic Web Road map," vol. Dec. 2004. Cambridge. MA: World Wide Web Consortium, 1998.
[92] J. Ferber, *Multi-agent systems: an introduction to distributed artificial intelligence*. Harlow; New York: Addison-Wesley, 1999.
[93] R. L. Hobbs, "Using XML to Support Military Decision-Making," presented at XML Conference and Exposition 2003, Philadelphia, PA, 2003.

[94] D. Fensel, *Spinning the semantic Web: bringing the World Wide Web to its full potential*. Cambridge, Mass.: MIT Press, 2003.
[95] A. Church, *Introduction to mathematical logic*. Princeton: Princeton University Press, 1956.
[96] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. New York: Addison-Wesley Pub, 2000.
[97] R. Carnap and R. Carnap, *Introduction to semantics, and Formalization of logic*. Cambridge: Harvard University Press, 1961.
[98] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*. Cambridge, Mass.: MIT Press, 1990.
[99] J. Sturdy, "A Lisp through the looking glass," University of Bath, 1991.
[100] A. J. T. Davie, *An introduction to functional programming systems using Haskell*. Cambridge; New York, NY, USA: Cambridge University Press, 1992.
[101] C. Chomsky, *The acquisition of syntax in children from 5 to 10*. Cambridge, Mass.: M.I.T. Press, 1969.
[102] N. Chomsky, *Aspects of the theory of syntax*. Cambridge: M.I.T. Press, 1965.
[103] G. Lakoff and M. Johnson, *Metaphors we live by*. Chicago: University of Chicago Press, 1980.
[104] G. Lakoff and R. E. Nunez, *Where mathematics comes from: how the embodied mind brings mathematics into being*. New York: Basic Books, 2000.
[105] H. D. Jørgensen, "Interactive Process Models," in *Department of Computer and Information Sciences*, vol. Ph. D.: Norwegian University of Science and Technology, 2003, pp. 304.
[106] H. A. Simon, *The sciences of the artificial*, 3rd ed. Cambridge, Mass.: MIT Press, 1996.
[107] J. Larkin and H. Simon, " Why a diagram is (sometimes) worth ten thousand words," *Cognitive Science*, vol. 11, pp. 65-99, 1987.
[108] T. Lindholm and F. Yellin, *The Java virtual machine specification*. Reading, Mass.: Addison-Wesley, 1997.
[109] M. C. Daconta, L. J. Obrst, and K. T. Smith, *The Semantic Web: a guide to the future of XML, Web services, and knowledge management*. Indianapolis, Ind.: Wiley Pub., 2003.
[110] D. Howe, "Free On-Line Dictionary of Computing," Imperial College Department of Computing, 2004.
[111] D. Lea, *Concurrent programming in Java: design principles and patterns*. Reading, Mass.: Addison Wesley, 1997.
[112] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Communications of ACM*, vol. 9, pp. 366-371, 1966.
[113] A. Goldberg and D. Robson, *Smalltalk-80: the language*. Reading, Mass.: Addison-Wesley, 1989.
[114] K. Jensen, *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Berlin; New York: Springer-Verlag, 1992.
[115] G. F. Cozman, "JavaBayes version 0.346," vol. 2004, 2004.
[116] B. B. Bederson, J. Grosjean, and J. Meyer, "Toolkit Design for Interactive Structured Graphics," CiteSeer, 2004.
[117] S. Pedroni and N. Rapping, *Jython essentials*. Sebastopol, Calif.: O'Reilly, 2002.
[118] C. G. Brooks, J. M. Grimwood, and L. S. Swenson, *Chariots for Apollo: a history of manned lunar spacecraft*. Washington: Scientific and Technical Information Branch, National Aeronautics and Space Administration, 1979.
[119] C. R. Pellegrino and J. Stoff, *Chariots for Apollo: the making of the lunar module*, 1st ed. New York: Atheneum, 1985.
[120] D. Joyner, *Adventures in group theory: Rubik's Cube, Merlin's machine, and other mathematical toys*. Baltimore, Md.: Johns Hopkins University Press, 2002.
[121] E. Frazzoli, *"Robust hybrid control for autonomous vehicle motion planning,"* Ph. D. :Massachusetts Institute of Technology. Dept. of Aeronautics and Astronautics., 2001, pp. 150 p.
[122] C. G. Brooks, J. M. Grimwood, and J. Loyd S. Swenson, "Chariots for Apollo: A History of Manned Lunar Spacecraft," NASA, 1979.
[123] G.-Q. Zhang, *Domain theory, logic, and computation: proceedings of the 2nd International Symposium on Domain Theory, Sichuan, China, October 2001*. Dordrecht; Boston: Kluwer Academic, 2004.
[124] J. Houbolt, "Manned Lunar-Landing through use of Lunar-Orbit," vol. 1, NASA, Ed.: NASA, 1961, pp. 99.
[125] J. Hansen, "Enchanted Rendezvous: John Houbolt and the Genesis of the Lunar Orbit Rendezvous Concept," NASA, 1999.
[126] D. Wallace, "Integrated Product Design Simulation," vol. 2005: MIT CADlab, 2002.

[127] NASA, "The Vision for Space Exploration," vol. 2005: NASA, 2004.
[128] D. H. Gelernter, *Mirror worlds, or, The day software puts the universe in a shoebox--: how it will happen and what it will mean*. New York: Oxford University Press, 1991.
[129] H. A. Simon, *Models of bounded rationality*. Cambridge, Mass.: MIT Press, 1982.
[130] D. J. Watts, *Small worlds: the dynamics of networks between order and randomness*. Princeton, N.J.: Princeton University Press, 1999.
[131] A. L. Barabasi, "The physics of the Web," *Physics World*, vol. 14, pp. 33-38, 2001.
[132] A. L. Barabasi, R. Albert, and H. Jeong, "Scale-free characteristics of random networks: the topology of the World-Wide Web," *Physica A*, vol. 281, pp. 69-77, 2000.
[133] A. L. Barabasi, E. Ravasz, and T. Vicsek, "Deterministic scale-free networks," *Physica A*, vol. 299, pp. 559-564, 2001.
[134] J. Doyle and J. M. Carlson, "Power laws, highly optimized tolerance, and generalized source coding," *Physical Review Letters*, vol. 84, pp. 5656-5659, 2000.
[135] B. Meyer, *Object-oriented software construction*. New York: Prentice-Hall, 1988.
[136] S. A. Whitmire, *Object-oriented design measurement*. New York: Wiley Computer Pub., 1997.
[137] M. Abadi and L. Cardelli, *A theory of objects*. New York: Springer, 1996.
[138] J. L. Hennessy, D. A. Patterson, D. Goldberg, and K. Asanovi, *Computer architecture: a quantitative approach*, 3rd ed. San Francisco: Morgan Kaufmann Publishers, 2003.
[139] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, N.J.: Prentice Hall, 1996.
[140] L. v. Bertalanffy, *General system theory: foundations, development, applications*. Harmondsworth: Penguin, 1973.
[141] A. H. M. t. Hofstede, E. Lippe, and T. P. v. d. Weide, "*A Categorical Framework for Conceptual Data Modeling: Definition, Application, and Implementation*," in *Technical Report*. Nijmegen: Computing Science Institute, University of Nijmegen, 1995.
[142] A. H. M. t. Hofstede, H. A. Proper, and T. P. v. d. Weide, "*Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models*," in *Information Systems*, 1993.
[143] J. Pearl, *Causality: models, reasoning, and inference*. Cambridge, U.K.; New York: Cambridge University Press, 2000.