

12-2011

# A FORMAL REPRESENTATION OF MECHANICAL FUNCTIONS TO SUPPORT PHYSICS-BASED COMPUTATIONAL REASONING IN EARLY MECHANICAL DESIGN

Chiradeep Sen

Clemson University, [chiradeep.sen@gmail.com](mailto:chiradeep.sen@gmail.com)

Follow this and additional works at: [https://tigerprints.clemson.edu/all\\_dissertations](https://tigerprints.clemson.edu/all_dissertations)



Part of the [Mechanical Engineering Commons](#)

---

## Recommended Citation

Sen, Chiradeep, "A FORMAL REPRESENTATION OF MECHANICAL FUNCTIONS TO SUPPORT PHYSICS-BASED COMPUTATIONAL REASONING IN EARLY MECHANICAL DESIGN" (2011). *All Dissertations*. 826.

[https://tigerprints.clemson.edu/all\\_dissertations/826](https://tigerprints.clemson.edu/all_dissertations/826)

This Dissertation is brought to you for free and open access by the Dissertations at TigerPrints. It has been accepted for inclusion in All Dissertations by an authorized administrator of TigerPrints. For more information, please contact [kokeefe@clemson.edu](mailto:kokeefe@clemson.edu).

A FORMAL REPRESENTATION OF MECHANICAL FUNCTIONS  
TO SUPPORT PHYSICS-BASED COMPUTATIONAL REASONING  
IN EARLY MECHANICAL DESIGN

---

A Dissertation  
Presented to  
the Graduate School of  
Clemson University

---

In Partial Fulfillment  
of the Requirements for the Degree  
Doctor of Philosophy  
Mechanical Engineering

---

by  
Chiradeep Sen  
December 2011

---

Accepted by:  
Dr. Joshua D. Summers, Committee Chair  
Dr. Gregory M. Mocko, Committee Co-Chair  
Dr. Georges M. Fadel  
Dr. Joel S. Greenstein

## ABSTRACT

The lack of computational support to the conceptual phase of mechanical engineering design is well recognized. Function-based modeling and thinking is widely recommended in design texts as useful means for describing design concepts and using them in tasks such as solution search, problem decomposition, and design archival. Graph-based function structure models that describe a product as a network of transformative actions of material, energy, and information, are discussed as a potential tool for this purpose, but in the current state of the art, function structures are not formalized as a computational representation. Consequently, no computer tool exists with which a designer can construct grammatically controlled function structure models, explore design ideas by model editing, and perform automated reasoning on the model against the laws of nature to draw analytical inferences on the design. This research presents, verifies, and validates a formal representation of mechanical functions that supports consistent computer-aided modeling of early design and reasoning on those models based on two universal principles of physics: (1) conservation and (2) irreversibility. The representation is complete in three layers. The first layer—the Conservation Layer—is defined with nine entities, five relations, five attributes, and 33 grammar rules that together formalize the construction of function structure graphs and support conservation-based qualitative validation of design concepts. The second layer—the Irreversibility Layer—includes three additional attributes that support both conservation-based and irreversibility-based reasoning at qualitative and quantitative

levels. The third layer—the Semantic Layer—is an extension of the previous two, where a vocabulary of nine verbs that describe mechanical devices and physical principles as functions is proposed. This layer supports feature-based modeling and semantic reasoning of function structures. The internal consistency of the representation is verified by logical examination and ontological consistency checking using Protégé-OWL. The coverage of the verbs is examined by constructing descriptive function structure models of a variety of existing physical principles and devices. The research is validated by incorporating the representation in a software tool using an object-oriented language and graphic user-interface, and by using the tool to construct models and demonstrate conservation-based and irreversibility-based reasoning.

## DEDICATION

To Sunrita and Ishaan

## ACKNOWLEDGEMENT

It is hard to express in words how much my association with my advisors—Dr. Joshua D. Summer and Dr. Gregory M. Mocko—over the past five years at Clemson has influenced my overall attitude and views toward academic research, rational thinking, and life as a whole. I want to take this opportunity to especially thank them for teaching me how to do research, for pushing me when I slacked, for helping me with my thoughts with their criticism and advice, and for being truly dependable friends. I owe much of my academic and professional growth directly to them. I also thank Dr. Georges M. Fadel and Dr. Joel S. Greenstein, my advisory committee members, for helping me through the review of this dissertation and asking me questions that prepared me better to address the research presented here. A special note of thanks and gratitude is for Dr. Jean-Marc Delhaye, especially for guiding me through some critical components of this work.

I thank the fellow graduate students of the Clemson Engineering Design Applications and Research Lab (CEDAR) for their friendship, support, and for continuously enriching my thoughts with alternate viewpoints. I specifically thank Ben Caldwell and Beshoy Morkos—fellow doctoral students that shared many reflections with me along this growth path called graduate school. I thank Prabhu Shankar, Essam Namouz, Jesse Schultz, Carl Lamar, Parikshit Mehta, and Shraddha Joshi for being supportive and critical of my work through my time at Clemson. I also thank my

students in the ME Senior Design program at Clemson for giving me the opportunities to consolidate my ideas through teaching.

Finally, I would not disgrace with mere words of thanks the contributions and sacrifice that my family—especially my wife Sunrita and son Ishaan—have made to support my plans of pursuing doctoral studies in mid-career. These are the two people that directly endured the brunt of the emotional and financial duress that came along with this project and who made uncountable little sacrifices everyday through these years to make my doctoral degree a reality. I could not have personally endured the stresses of this program without the love of Sunrita and Ishaan, and the emotional support from my parents, Arup and Reba, and my sister, Paramita. I am indebted to all these people for life. I also want to thank Mr. Alok Roy Choudhury and his family for their friendship, love, and support. Last, but not the least, I thank my neighbors and friends—Rajan and Priya—for providing me and my family with residential accommodation in the final weeks of finishing this dissertation. Without all and each of these people, this work would not be possible.

## TABLE OF CONTENTS

Abstract.....	ii
Dedication.....	iv
Acknowledgement.....	v
List of Figures.....	xv
List of Tables.....	xxi
List of Model States.....	xxv
List of Algorithms.....	xxvi
Intentionally Left Blank.....	xxvii
Chapter 1. Research Overview.....	1
1.1 Motivation: Why Create Physics-Based Function Models?.....	1
1.2 Core Concepts and Definitions.....	4
1.2.1 Representation.....	4
1.2.2 Formal Representation.....	5
1.2.3 Model.....	5
1.2.4 Modeling.....	6
1.2.5 Reasoning.....	7
1.2.6 Flow and Flow Noun.....	7
1.2.7 Function and Function Verb.....	9



1.2.8	Function Structure.....	10
1.2.9	Topology.....	13
1.3	Summary of High-Level Requirements for the Representation.....	14
1.4	Research Questions, Hypotheses, and Tasks .....	16
1.5	Solution Overview.....	20
Chapter 2.	Review of Function-Based Design .....	23
2.1	The AI Views of Function-Based Design .....	23
2.2	The Engineering Design View of Function-Based Design .....	27
2.3	Comparison of the Two Views.....	31
2.4	The Functional Basis and the Design Repository .....	32
2.5	Research Gap Analysis: Lack of Rigor in the Function Structure Formalism... 39	
2.5.1	Vocabulary-Level Discrepancies .....	40
2.5.2	Definition-Level Lack of Rigor .....	42
2.5.3	Model-Level Discrepancies .....	47
2.6	Modeling Flexibility and Expressive Power of Notional Terms.....	50
Chapter 3.	Requirements Analysis for the Representation.....	53
3.1	Coverage over Multiple Physics Domains .....	54
3.1.1	What is Coverage over Physics Domains? .....	54
3.1.2	Coverage of Contemporary Representations .....	55
3.1.3	The Coverage Requirement .....	56
3.2	Domain-Independence of Physics Laws .....	56
3.2.1	What is Domain-Dependence? .....	56

3.2.2	Level of Domain-Dependence of Comparable Representations.....	59
3.2.3	The Domain-Independence Requirement .....	62
3.3	Physics-Based Concreteness of Modeling Terms .....	63
3.3.1	What is Concreteness? .....	63
3.3.2	Concreteness of Existing Function Vocabularies .....	65
3.3.3	The Physics-Based Concreteness Requirement.....	70
3.4	Normative and Descriptive Modeling Support .....	71
3.4.1.1	What are Normative and Descriptive Models?.....	71
3.4.2	Characterization of Existing Function Representations.....	73
3.4.3	The Normative and Descriptive Modeling Requirement.....	74
3.5	Qualitative Modeling and Reasoning Support .....	75
3.5.1	What is a Qualitative Function Model? .....	75
3.5.2	What is Physics-Based Qualitative Reasoning? .....	75
3.5.3	Qualitative Modeling and Reasoning in Contemporary Representations...	77
3.5.4	The Qualitative Modeling Requirement .....	78
3.5.5	The Qualitative Reasoning Requirement .....	79
3.6	Extendibility .....	79
3.6.1	What is Extendibility?.....	79
3.6.2	Quantitative Reasoning Extension.....	80
3.6.3	Causal Reasoning Extension.....	81
3.7	Scalability.....	82
3.7.1	What is Scalability? .....	82

3.7.2	The Scalability Requirement.....	83
3.8	Consistency and Validity .....	85
Chapter 4.	Systematic Discovery of Reasoning Needs and Information Elements for the Representation.....	87
4.1	Discovering Reasoning Needs: The Chalkboard Exercise.....	87
4.1.1	Design Problem Selection.....	87
4.1.2	Participant Selection .....	89
4.1.3	Modeling Interface and Feedback.....	90
4.1.4	Exercise Steps: Black Box Modeling (Qualitative Reasoning) .....	91
4.1.5	Exercise Steps: Model Decomposition (Quantitative Reasoning).....	101
4.2	Reasoning Algorithms.....	112
4.2.1	Conservation Reasoning Algorithms (Topologic) .....	112
4.2.2	Conservation Reasoning Algorithms (Derivational) .....	114
4.2.3	Irreversibility Reasoning Algorithm.....	119
4.2.4	Quantitative Reasoning Algorithm (Power Required).....	120
4.3	Information Elements Extraction .....	126
Chapter 5.	Representation Layer One: Formalization of Function Structure for Conservation Reasoning .....	132
5.1	Layer 1 Vocabulary .....	133
5.1.1	Layer 1 Entity Types.....	136
5.1.2	Layer 1 Vocabulary of Relation Types.....	148
5.1.3	Layer 1 Vocabulary of Attribute Types and their Correspondence .....	150

5.2	Layer 1 Local Grammar .....	152
5.2.1	Unary Grammar Rules for Input-Output Relations (HeadNode, TailNode)...	
	.....	156
5.2.2	Unary Grammar Rules for Carrier-Carried Relations.....	158
5.2.3	Unary Grammar Rules for Parent-Child Relations.....	160
5.2.4	Binary Grammar Rules for Input-Output Relations.....	167
5.2.5	Binary Grammar Rules for Carrier-Carried Relations.....	172
5.2.6	Special Grammar Rules for Layer 1 .....	180
Chapter 6. Implementation and Validation: Modeling and Reasoning Demonstration		
	with Layer One .....	182
6.1	Demonstration of Internal Consistency.....	184
6.1.1	Logical Examination of Exhaustiveness of Local Grammar .....	185
6.1.2	Logical Examination of Consistency of Local Grammar .....	189
6.1.3	Ontological Examination of Consistency of the Vocabulary .....	195
6.2	Demonstration of External Validity against Conservation Laws.....	209
6.2.1	Design of the Software Tool ConMod.....	209
6.2.2	Demonstration of Function Modeling and Qualitative Conservation Reasoning .....	234
6.2.3	Application to Product-Level Modeling and Reasoning (Scalability).....	242
Chapter 7. Representation Layer Two: Extension of Layer One for Irreversibility-based Reasoning.....		
		247
7.1	Extension of the Representation to Include Irreversibility-Based Attributes ..	249

7.2	Implementation and Validation: Qualitative Irreversibility Reasoning .....	251
7.3	Implementation and Validation: Quantitative Irreversibility Reasoning .....	262
Chapter 8. Representation Layer Three: Semantic Layer: A Physics-Based Vocabulary of Function Verbs .....		
		279
8.1	The Need for Semantic Information in Function Modeling and Reasoning ....	280
8.2	Proposed Vocabulary of Atomic Function Verbs .....	287
8.2.1	Energy Verbs – Primary Level .....	289
8.2.2	Energy Verbs – Secondary Level .....	298
8.2.3	Material Verbs .....	311
8.2.4	Topologic Verbs.....	315
Chapter 9. Validation of Layer Three: Modeling Coverage of the Physics-Based Verbs.....		
		322
9.1	Coverage Testing of Energy Verbs through Descriptive Modeling (Closed Systems) .....	322
9.1.1	Storage and Supply of Electrical Energy (Device: Lead-Acid Battery) ...	323
9.1.2	Resistance to Electrical Current (Devices: Resistor, Heating Coil, Lamp Filament).....	326
9.1.3	Storage and Supply of Electrical Energy using Capacitance (Device: Capacitor) .....	330
9.1.4	Production of Magnetic Field using Inductance (Device: Solenoid with or without Soft Iron Slider).....	332

9.1.5	Work from Electrical Energy (Device: DC Motor with Permanent Magnet).....	335
9.1.6	Work from Electrical Energy (Device: DC Motor with Field Winding)..	337
9.1.7	Electrical Energy from Work (Device: DC Generator with Permanent Magnet).....	339
9.2	Coverage Testing of Material Verbs through Descriptive Modeling (Open Systems) .....	341
9.2.1	Heat Transfer between Two fluids across a Wall (Device: Heat Exchanger Pipe).....	341
9.2.2	Heat Transfer from a Fluid to the Atmosphere (Device: Radiator) .....	342
9.2.3	Heat Transfer through an Intermediate Cycled Flow (Device: Disk Heat Exchanger).....	343
9.2.4	Free Drainage of Water from a Tank (Device: Penstock of a Hydraulic Turbine).....	345
9.2.5	Conversion of Kinetic Energy of Water to Shaft Work (Device: Francis Turbine).....	347
9.3	Extension of Physics-Based Verbs with Residual Flows for Feature-Based Modeling .....	350
9.4	Product-Level Coverage: Descriptive Models from the Design Repository....	357
9.4.1	The Hairdryer Function Structure .....	357
9.4.2	The Shop-Vac Function Structure .....	363
9.5	Product-Level Coverage: Normative Modeling of a New Product Concept ...	371

Chapter 10. Closure, Ongoing Work, and Path Forward.....	377
10.1 Overall Research Outcome.....	377
10.2 Contributions to the State of the Art .....	378
10.3 Evaluating the Representation against the High-Level Requirements.....	382
10.4 Answers to Research Questions and Hypotheses.....	387
10.5 Ongoing Extensions and Future Research Directions.....	394
10.5.1 Ongoing Work: Formalization of Notional Verbs .....	395
10.5.2 Ongoing Work: Examination of Designer-Level Usability .....	404
10.5.3 Future Work: Causal Reasoning Extension .....	409
10.5.4 Future Work: Additional Directions to Explore .....	412
Appendices.....	414
Appendix A. Reasoning Discovery Experiment Steps Discussed in Section 4.1 ...	415
Appendix B. XML Code for the Function Ontology Presented in Section 6.1.3 ...	439
Appendix C. Header Files for the ConMod Application .....	457
Appendix D. Source Files for the ConMod Application.....	488
References.....	610

## LIST OF FIGURES

Figure 1.1: Examples of mechanical functions.....	9
Figure 1.2: Function structure model of a hairdryer stored in the Design Repository .....	11
Figure 1.3: Conventional and bipartite views of a function structure model.....	13
Figure 2.1: The transformative view of device function [1].....	27
Figure 2.2: Function structure of a hairdryer stored in the Design Repository .....	28
Figure 2.3: Sample graph grammar rule for function model synthesis (Redrawn from [6]).....	31
Figure 2.4: Function structure of an electrical motor using the Functional Basis (secondary level).....	35
Figure 2.5: Definitions of terms within the Design Repository.....	36
Figure 2.6: Artifact browser in the Design Repository showing the heating coil frame of the hairdryer.....	38
Figure 2.7: Logical Definition of Channel derived by taking intersection of the definitions in Table 2.5 .....	47
Figure 2.8: Illustration of model-level inconsistencies.....	48
Figure 3.1: Loss of generality of physics equations due to increasing domain-dependence (Adapted from [126]).....	58
Figure 3.2: Expression of hydrostatic pressure in terms of universal gravitation .....	59
Figure 3.3: Use of the verb Separate in two different models in the Design Repository .	66
Figure 3.4: Different physics of the same verb in two applications .....	68



Figure 3.5: A typical function model for energy conversion.....	75
Figure 3.6: Qualitative and quantitative models.....	80
Figure 3.7: Subgraph being searched within a function model .....	83
Figure 4.1: A sample step from the chalkboard modeling exercise.....	92
Figure 5.1: Entity-relation-attribute (ERA) model for the Level 1 vocabulary.....	135
Figure 5.2: Redundant topologic data elements not captured in any class .....	143
Figure 5.3: Assignment of carrier flow, head node, and tail node based on flow type ..	144
Figure 5.4: An internally inconsistent model of a heat exchange function .....	152
Figure 5.5: The consistent construct of the heat exchange function.....	155
Figure 6.1: OWL class hierarchy (Asserted) .....	197
Figure 6.2: Exclusive disjunction between Material, Energy, and Signal (Asserted) ....	198
Figure 6.3: Object properties and data properties (Asserted) .....	200
Figure 6.4: Domain, range, and other characteristics of properties (Asserted) .....	202
Figure 6.5: Restrictions on Energy (Asserted and Inferred).....	204
Figure 6.6: Consistency checking results for the ontology (Consistent) .....	205
Figure 6.7: Consistency checking results for the ontology (Consistent) .....	206
Figure 6.8: OWL class hierarchy (Inferred) – Identical with the asserted hierarchy .....	206
Figure 6.9: Creation of individual instances using the ontology .....	208
Figure 6.10: The document-view architecture of ConMod .....	211
Figure 6.11: Class diagram of the ConMod application.....	214
Figure 6.12: ConMod main window and toolbar buttons (Layer One) .....	231
Figure 6.13: Dialog boxes for adding function and environment instances (Layer One)	232

Figure 6.14: Dialog boxes for adding Material, Energy, and Signal instances (Layer One).....	233
Figure 6.15: Qualitative conservation reasoning message dialog.....	234
Figure 6.16: Model for topologic reasoning and derivational reasoning (# 1 – 5).....	237
Figure 6.17: Derivational reasoning output from the model in Figure 6.16.....	238
Figure 6.18: Model for derivational reasoning # 6, 7, and 9 .....	240
Figure 6.19: Derivational reasoning output from the model in Figure 6.18.....	240
Figure 6.20: Model for derivational reasoning # 9a .....	241
Figure 6.21: Derivational reasoning output from the model in Figure 6.20.....	242
Figure 6.22: The Air Heater model from Model State 4.14 reconstructed in ConMod..	244
Figure 6.23: Reasoning output for the Air Heater model in Model State 4.14.....	245
Figure 7.1: Energy loss in physical processes .....	248
Figure 7.2: Extension of the representation to support irreversibility reasoning.....	251
Figure 7.3: Dialog boxes for adding Material and Energy instances (Layer Two - Qualitative) .....	253
Figure 7.4: Reasoning menu options in ConMod (Layer Two - Qualitative).....	253
Figure 7.5: Air Heater model of Model State 4.14 reconstructed using ConMod (Layer 2).....	255
Figure 7.6: Qualitative derivational reasoning produced by ConMod-2 on the model shown in Figure 7.5.....	256
Figure 7.7: Qualitative irreversibility report produced by ConMod-2 on the model shown in Figure 7.5 .....	257

Figure 7.8: Modified model of the Air Heater, with residual flows identified by modeler .....	258
Figure 7.9: Qualitative irreversibility report produced by ConMod-2 on the model shown in Figure 7.8 (modified model with some residual flows marked).....	259
Figure 7.10: Depiction of residual flows for different design intent and zero magnitude.....	262
Figure 7.11: Add Energy dialog and reasoning toolbar (Layer Two - Quantitative) .....	263
Figure 7.12: Reasoning menu options in ConMod (Layer Two - Quantitative).....	263
Figure 7.13: Quantitative model of the Air Heater using ConMod-2q, showing default power of Energy flows and one negative power value .....	266
Figure 7.14: Quantitative reasoning Step-1: Check for negative power magnitudes .....	267
Figure 7.15: Aborting reasoning steps under out-of-date model state.....	267
Figure 7.16: Quantitative model of the Air Heater using ConMod-2q, after correcting negative power values.....	269
Figure 7.17: Quantitative reasoning Step-2: Check for quantitative energy balance .....	270
Figure 7.18: Air Heater model after manually ensured energy balance .....	273
Figure 7.19: Quantitative energy balance report: Step 2 with passing results.....	274
Figure 7.20: Quantitative reasoning Step-3: Computing function-wise and model efficiency.....	275
Figure 8.1: Lack of formalism to capture function semantics .....	281
Figure 8.2: Semantic inconsistency between function description and topology .....	282
Figure 8.3: An instance of the proposed verb Energize_M .....	285

Figure 8.4: Two possible return values for solution search .....	287
Figure 9.1: Storage of electrical energy as chemical potential energy .....	324
Figure 9.2: Supply of electrical energy from stored chemical potential energy .....	325
Figure 9.3: Resistive heating.....	327
Figure 9.4: Storage of electrical energy as electrical potential energy (static charge) ...	330
Figure 9.5: Supply of electrical work (current) from stored electrical potential energy	331
Figure 9.6: Production of magnetic field without mechanical work by induction .....	333
Figure 9.7: Production of magnetic force from electric energy through induction .....	334
Figure 9.8: DC motor with permanent magnet.....	335
Figure 9.9: DC motor with field winding .....	338
Figure 9.10: DC motor model with energy transfer functions.....	339
Figure 9.11: DC generator with permanent magnet.....	340
Figure 9.12: Heat exchange between two fluids across a wall .....	342
Figure 9.13: Heat transfer from hot fluid in a pipe to atmospheric air .....	343
Figure 9.14: Heat transfer using an intermediate reused flow .....	344
Figure 9.15: Schematic diagram of free drainage of water from a tank .....	346
Figure 9.16: Conversion of potential energy into kinetic energy in free drainage of liquids.....	347
Figure 9.17: Extraction of shaft work from kinetic energy of water .....	348
Figure 9.18: Geometric CAD model of a boss feature .....	351
Figure 9.19: Function structure model of a hairdryer stored in the Design Repository .	358
Figure 9.20: The hairdryer function structure using the physics-based verb primitives	359

Figure 9.21: The hairdryer function structure using the physics-based verb features ....	366
Figure 9.22: Shop-Vac function model within the Design Repository.....	367
Figure 9.23: Shop-vac function structure using the physics-based verb primitives .....	368
Figure 9.24: Shop-Vac function structure constructed using the physics-based verb features.....	369
Figure 9.25: Normative model of an automatic omelet maker .....	373
Figure 10.1: Protocol to formalize notional verbs (In-progress future work).....	396
Figure 10.2: Syntactic translation of the existing textual definition of Branch available in Functional Basis literature [26].....	397
Figure 10.3: Incorrect use of the verb Branch allowed by its definition .....	398
Figure 10.4: Intended usage of the verb Branch not supported by its definition.....	398
Figure 10.5: Partial branching of mixtures not allowed by the definition of Branch .....	399
Figure 10.6: Functions conflicting with natural laws are supported by the definition ...	399
Figure 10.7: Energy exchange between the system and surrounding cannot be modeled.....	401
Figure 10.8: Formalized definition of Branch consistent with the conservation laws....	403
Figure 10.9: Sample results of pilot protocol studies on designer-model interaction ....	405
Figure 10.10: Activity Encoding for Participant P1 .....	406
Figure 10.11: Element encoding for participant P1 .....	407
Figure 10.12: Function structure of an electric motor and pump assembly.....	409
Figure 10.13: Causal relationships to be captured in the extended representation.....	410
Figure 10.14: Causal Reasoning Tree.....	411

## LIST OF TABLES

Table 1.1: Research Question 1, hypotheses, and tasks.....	17
Table 1.2: Research Question 2, hypotheses, and tasks.....	19
Table 2.1: The Functional Basis Verb Set .....	33
Table 2.2: The Functional Basis Nouns Set.....	34
Table 2.3: Energy flow types within the Functional Basis .....	42
Table 2.4: Sample verbs and notional definitions from the Functional Basis [26].....	43
Table 2.5: First order logic-based translation of four Functional Basis verbs.....	46
Table 4.1: Redundant function inference.....	93
Table 4.2: Dangling tail and barren flow inference .....	93
Table 4.3: Dangling head and orphan flow inference.....	94
Table 4.4: Material transformation without energy exchange .....	95
Table 4.5: Environments as singularity nodes .....	96
Table 4.6: Unused (barren) energy flow .....	97
Table 4.7: Carrier flow and irreversibility inference .....	98
Table 4.8: One-in-many-out derivation (acceptable black box model) .....	100
Table 4.9: Topologic inferences during decomposition .....	102
Table 4.10: Material transformation without energy in a decomposed model .....	102
Table 4.11: Many-in-one-out inference (Accepted model) .....	104
Table 4.12: Flow preservation and additive inference in decomposition (Draw Air) ....	105
Table 4.13: Flow preservation inference in decomposition (Deliver Air).....	107

Table 4.14: Quantitative reasoning on efficiency and power required .....	108
Table 4.15: Summary of Reasoning Needs Discovered .....	110
Table 4.16: Extraction of information elements from the algorithms .....	127
Table 5.1: Layer 1 vocabulary .....	134
Table 5.2: Layer 1 entity: Element .....	137
Table 5.3: Layer 1 Entity: Node .....	138
Table 5.4: Layer 1 entity: Noun .....	139
Table 5.5: Layer 1 entity: Verb .....	140
Table 5.6: Layer 1 entity: Environment .....	141
Table 5.7: Layer 1 entity: Source .....	141
Table 5.8: Layer 1 entity: Sink .....	142
Table 5.9: Layer 1 entity: Material .....	145
Table 5.10: Layer 1 entity: Energy .....	146
Table 5.11: Layer 1 entity: Signal .....	147
Table 5.12: Layer 1 relation types and descriptions .....	148
Table 5.13: Layer 1 attributes and descriptions .....	150
Table 5.14: Layer 1 grammar rules: Unary input-output relations .....	156
Table 5.15: Constructs controlled by the unary input-output rules .....	157
Table 5.16: Layer 1 grammar rules: Unary carrier-carried relations .....	158
Table 5.17: Constructs controlled by the unary carrier-carried rules .....	160
Table 5.18: Layer 1 grammar: Unary parent-child relations .....	161
Table 5.19: Constructs controlled by the unary parent-child rules .....	162

Table 5.20: Constructs controlled by the unary parent-child rules (Rule 12).....	167
Table 5.21: Layer 1 grammar: Binary input-output relations for flows .....	168
Table 5.22: Constructs controlled by the binary input-output rules .....	169
Table 5.23: Layer 1 grammar: Binary input-output relations for nodes.....	171
Table 5.24: Constructs controlled by the binary input-output rules .....	171
Table 5.25: Layer 1 grammar: Binary carrier-carried relations.....	172
Table 5.26: Constructs controlled by the binary carrier-carried rules .....	173
Table 5.27: Layer 1 grammar: Special rule .....	180
Table 6.1: Domain, math_func, and range for unary input-output rules .....	190
Table 6.2: Domain, math_func, and range for all local grammar rules .....	191
Table 6.3: Class CGeometry .....	215
Table 6.4: Class CElement (Element).....	216
Table 6.5: Class CNode (Node).....	218
Table 6.6: Class CEdge (Noun) .....	219
Table 6.7: Class CFunction (Verb).....	221
Table 6.8: Class CEnv (Environment).....	222
Table 6.9: Class CMaterial (Material) .....	223
Table 6.10: Class CEnergy (Energy) .....	224
Table 6.11: Class CSignal (Signal).....	224
Table 6.12: Class CConModDoc (Document).....	225
Table 6.13: Class CConModView (View).....	226
Table 6.14: Conservation reasoning to be validated using ConMod .....	235



Table 7.1: Layer 2 attributes and descriptions .....	249
Table 8.1: Primary energy verb: TypeChange_E .....	290
Table 8.2: Primary energy verb: Transfer_E .....	294
Table 8.3: Primary energy verb: Change_E.....	295
Table 8.4: Primary energy verb: Store_E .....	297
Table 8.5: Primary energy verb: Supply_E.....	298
Table 8.6: Three modes of transfer and storage for different energy types .....	302
Table 8.7: Primary energy verb: Conduct_E .....	308
Table 8.8: Primary energy verb: Radiate_E.....	309
Table 8.9: Summary of energy verbs and their description tables.....	310
Table 8.10: Material verb: Energize_M.....	312
Table 8.11: Material verb: DeEnergize_M.....	314
Table 8.12: Topologic verb: Logical_Branch.....	316
Table 8.13: Topologic verb: Logical_Unite.....	318
Table 8.14: The proposed physics-based verbs and their graphical symbols .....	319
Table 9.1: Model variation with design intent .....	329
Table 9.2: Extension of physics-based verbs with provisions for typical residual flows	352
Table 9.3: Function-name mapping between models (hairdryer).....	361
Table 9.4: Function-name mapping between models (Shop-Vac).....	363
Table 10.1: Answers to Research Questions and Hypotheses .....	387

## LIST OF MODEL STATES

Model State 4.1: Function instance AHD (Air Heating Device) .....	93
Model State 4.2: AHD with one input flow instance Air1 (type: Gas).....	93
Model State 4.3: AHD with one output flow instance Air2 (type: Gas).....	94
Model State 4.4: AHD with two material flows .....	95
Model State 4.5: Flow instances connected to environment instances.....	96
Model State 4.6: Energy flow EE1 input to support material transformation .....	97
Model State 4.7: EE1 transformed into ThE1 and added to Air2.....	98
Model State 4.8: Lost energy included in model: An acceptable model .....	100
Model State 4.9: User-driven decomposition (first level) .....	102
Model State 4.10: Stream of air flow through functions .....	102
Model State 4.11: Accepted model of first level decomposition.....	104
Model State 4.12: Second level decomposition of Draw Air .....	105
Model State 4.13: Second level decomposition of Deliver Air .....	107
Model State 4.14: Final Model State of the Exercise .....	108

## LIST OF ALGORITHMS

Algorithm 4.1: Algorithm for redundant function .....	112
Algorithm 4.2: Algorithm for dangling tail .....	113
Algorithm 4.3: Algorithm for dangling head.....	113
Algorithm 4.4: Algorithm for barren flow .....	115
Algorithm 4.5: Algorithm for orphan flow .....	115
Algorithm 4.6: Algorithm for one-in-many-out derivation inference .....	116
Algorithm 4.7: Algorithm for many-in-one-out derivation inference .....	117
Algorithm 4.8: Algorithm for material transformation without energy.....	119
Algorithm 4.9: Algorithm for qualitative detection of missing residual flow .....	119
Algorithm 4.10: Algorithm for computing power required .....	120
Algorithm 4.11: Algorithm for boundary flow preservation in decomposition.....	121
Algorithm 4.12: Algorithm for additive inference across decomposition levels.....	124

INTENTIONALLY LEFT BLANK

## CHAPTER 1. RESEARCH OVERVIEW

The objective of this dissertation research is to develop a formal representation of mechanical functions based on the governing physics of mechanical systems, specifically to meet the modeling and computational reasoning needs in early design. This chapter presents a high-level view of the overall research problem, the requirements of the solution, summary of the research questions, hypotheses, and tasks, and an overview of the solution, without providing details of the solution, its rationale, or validation. For each item mentioned above, a pointer to the section or chapter within the document where the item is addressed in greater detail is provided. After building motivation, some core concepts necessary to understand this research are defined. High level requirements that a system must satisfy to solve the overall problem are described next. The research answers two overarching research questions, which are presented along with their hypotheses and tasks. Finally, the representation is completed in three layers, as presented in the last section of the chapter.

### **1.1 Motivation: Why Create Physics-Based Function Models?**

The focus of early design lies on synthesis tasks such as ideation and concept generation [1-4] and much research has been directed toward automating design synthesis [5-13]. Traditionally, analysis is reserved for the later stages, where modeling and reasoning tools such as CAD, FEA, and CFD exist. While applicable to a broad range of problems, these tools typically need geometric and/or other quantitative information for

modeling and reasoning that may not exist in early novel design. Computer tools for early design analysis exist only in the evolutionary design of some complex systems such as automotive<sup>1</sup>, aerospace<sup>2</sup>, or nuclear power plants<sup>3</sup> [14], but they usually operate by reusing previously established domain-specific rules of reconfigurable subsystems. In the early stages of novel, open-ended mechanical design, solutions are synthesized often in abstract, qualitative, non-geometric form and a variety of physics principles may be considered. For these problems, general purpose tools that support modeling early, abstract design from a variety of physics domains, and perform suitably abstracted analysis, are needed. This analysis could include, but is not limited to, the theoretical correctness and feasibility of a concept, possible functional failure sources, failure propagation paths and effects, and the effects of changing a design parameter on the remainder of the design (design exploration). Such tools could not only enable electronic documentation of early design intent in dedicated file formats for viewing, editing, and evolving in subsequent product variant design cycles, but also help the designer to engage in a physics-based “conversation” with the model [15] while developing and exploring

---

<sup>1</sup> <https://www.modelica.org/>, accessed on August 16, 2011

<sup>2</sup> <http://www.pca2000.com/en/index.php>, accessed on August 16, 2011

<sup>3</sup> <http://www.intergraph.com/learnmore/ppm/power/nuclear-plant-construction.aspx>, accessed on August 16, 2011

design variants, and to examine the consequences of design decisions, while still within the early stage.

However, computational reasoning requires the necessary information elements to be available in a formal representation on which models can be constructed and reasoning algorithms can be executed. To this end, a function-based representation is developed in this dissertation. Function-based thinking is professed as a useful means to support early design thinking in design texts [1, 2]. To date, automation effort in this area is primarily channeled to aid design synthesis and many models exist to that end [5, 6, 16-22]. However, research shows that designers mentally archive and process their knowledge about engineering devices in terms of their functions and use function-based human reasoning to solve problems in early design [3, 23, 24]. This background makes function-based formal reasoning a strong candidate for automating early design analysis. Moreover, since functions can be modeled as graphs—a widely used and extendable data structure supported by rigorous mathematical basis—and since previous research made significant advances toward formalizing functions [16, 17, 25-29], it is anticipated that a function-based representation, when properly designed, would be suitable for supporting early design analysis reasoning. To facilitate easy interpretation of the remainder of this discussion, the core concepts of function-based modeling and formal representation are defined next.

## 1.2 Core Concepts and Definitions

Core concepts pertinent to this research are defined to support the discussion in the remainder of this chapter. In each definition sentence, the defined term is underlined.

### 1.2.1 Representation

The terms representation and model often are used interchangeably in literature [30, 31] and a distinction is necessary for this dissertation. Representation has been defined as a substitution of reality using a symbolism [31]. For this dissertation, a representation is the collective description of (1) a vocabulary of entity types, relation types, attributes, and (2) local grammar rules, which constitute the **general information structure** for describing specific members within a domain of discourse. An example is the boundary representation of three-dimensional geometry [32-34], which is used to define members (3-d objects) within the domain of discourse (3-d space), and includes a vocabulary of entity types (e.g., vertices, edges, faces, shells, holes, and loops), a vocabulary of relation types (e.g., boundary relations), and grammar rules to control the permitted combination of entity instances and relation instances (e.g., Euler's equation of manifold solid) [34]. This choice of information elements and their structure is generic for all instances of solid objects described using the boundary representation. Similarly, the vocabulary of symbols of electrical component types (entity types), electrical connection types (relation types), and guidelines for correctly combining instances of these entity types and relation types to describe an electrical device (grammar) constitutes a representation for drawing circuit diagrams. The representation is the generic information structure, as opposed to a specific solid or a specific circuit diagram, which



are models constructed on the respective representations. A classification of representations based on their vocabulary, structure, expression, purpose, and abstraction is proposed in previous research [31, 35-42].

### **1.2.2 Formal Representation**

A formal representation is a representation defined in a rigorous, computer-implementable form that describes how to store the entities, relations, and attributes in a computer data structure, render them on a computer screen, and support computational reasoning on them. The term formal implies that the information captured is stored in the syntactic “form” of the descriptions rather than in their semantics. Examples are the implementations of the boundary representation in solid geometry kernels such as ACIS [34], kernel-neutral CAD languages such as STEP AP-214 [43], and the Design Exemplar [44, 45].

### **1.2.3 Model**

A model is described as an abstraction of reality that can be used to answer some questions about that reality [46]. Distinguishing a model from its representation, a model is an instance of a specific member within a domain of discourse described by a representation. It consists of specific instances of the entities, relations, and modifiers defined within the representation. An example is a specific solid model of a machine part built using the boundary representation. This model substitutes the real part for purposes such as visualization and answering questions about its geometry, mass properties, fits, and tolerances. It describes a specific object using specific instances of vertices, edges,

faces, shells, holes, and loops, which are entities defined in the boundary representation. Similarly, the circuit diagram for a specific electrical device drawn in accordance with the set of rules described in the representation is a model. In this general sense, the word model includes any substitution of an entity with another, such as (1) the specific behavior equation of a spring-mass-damper system that answer questions about the position, velocity, and acceleration of the mass, (2) a specific scaled physical prototype of a building or an airplane that answers questions about its proportion, aerial view, or wind flow characteristics around it, or (3) a specific project plan that answers questions about activities on the critical path and an individual person's tasks for a day.

#### **1.2.4 Modeling**

Modeling is the activity of constructing a model to describe a specific object within a domain of discourse using a representation available to describe objects in that domain. It is important to identify modeling separately from model and representation since the final qualities of a model such as consistency, validity, completeness, and soundness, are results of those qualities in the representation itself and in the modeling action. Depending on the flexibility of the representation [31], it may be possible to construct an inconsistent model using an internally consistent representation and vice versa. For example, the term definitions in the Functional Basis vocabulary [25, 26, 47] are not formalized, as they are defined in natural English rather than a syntactic form, their flexibility is high. Consequently, they are often used in models that can be demonstrated as inconsistent [48]. With highly flexible representations, the onus of maintaining model-level consistency lies largely on the modeler, as the representation

does not enforce guidelines to that effect. One objective of this dissertation research is to formalize a function modeling representation such that the definitions restrict modeling flexibility to ensure conformity to the natural laws (Chapter 6), while not reducing the flexibility to model a wide range of physical principles and phenomena executed in mechanical systems (Chapter 9).

### **1.2.5 Reasoning**

Reasoning is the activity of using a model and logic described within the representation (local rules) or outside (global rules) to draw inferences about the model in order to derive information that is not explicitly captured within the model. An example is to derive the volume, surface area, or length of the diagonal of a rectangular block from its solid model that is created using its length, breadth, and height parameters. The sought information is not directly captured in the model's description, but can be derived from the model by using global rules that relate the three parameters available in the model to the sought parameters. Formal reasoning is the process of reasoning using a formal representation and algorithmic rules.

### **1.2.6 Flow and Flow Noun**

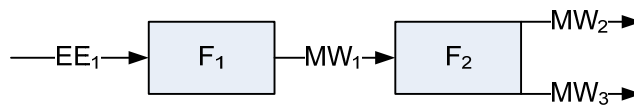
A flow is an occurrence of an entity type such as material, energy, or signal, which is either used or produced by an action performed by an artifact. It is distinguished from a fluid that is necessary for the operation of a device but is not used or produced by it. For example, in a closed thermodynamic system, such as gas enclosed in a cylinder, the gas is not a flow, since it is not entering or leaving the system. It *is* the system. Heat

and work that cross the boundary of the system are flows. Similarly, in an automobile cooling circuit studied as a system, the coolant fluid is not a flow, since it flows **within** the system, rather than **through** it. However, when the coolant pump and the radiator are studied as individual open systems, the coolant becomes a flow. Thus, whether a fluid is a flow or a system depends on the definition of the system.

The complete description of a flow includes its type, subtype (optional), and functional state. The functional state, or state, of a flow is the set of attributes and relations pertinent to its type and their values that distinguish the flow from other flows of the same type in a model. If two flows in a model have the same type, subtype, and functional state, it follows that they are the same instance and therefore, the model is redundant. In the proposed representation, a flow is characterized by the physical quantities applicable to its type (e.g., voltage and current, if the type is electrical energy), its Location attribute, and its carrier flows, controlled by the relation CarrierFlow. The first specifies a zone in the geometric space where the flow is identified and the second is a pointer to another flow that carries the flow in question. For example, an instance of Electrical Energy available at a specific wall socket, socket1, can be characterized by its type (Energy), subtype (Electrical Energy), and attributes voltage (110 V), current (1 A), and Location (socket1), which completes its description. Similarly, energy carried by water exiting a specific nozzle, nozzle1, in a Pelton turbine installation can be described as type (Energy), subtype (Kinetic Energy), mass flow rate (100 kg/s), Location (nozzle1), and CarrierFlow (pointer to the water flow instance). Two flows are defined

as different if they are of different types or if they are of the same type but at least one attribute value or relation value used to define their states is different between them.

For the purpose of formalizing, it is important to distinguish specific flow instances from the class Noun, from which those instances are derived. Flow Noun or Noun (proper noun, capitalized) is a class from which flow instance are derived and subclasses of more specific a flow types are inherited. The term “a noun” or “nouns” refers to one or more of these classes that inherit the Noun class. The term “a flow” or “flows” refer to instances of nouns. For example, in Figure 1.1, EE1 is a flow, which is an instance of the class Electrical Energy, and MW1 is another flow, which is an instance of the class Mechanical Work. These two classes are inherited from the class Noun.



**Figure 1.1: Examples of mechanical functions**

### 1.2.7 Function and Function Verb

A mechanical function, or a function, is an occurrence of a transformative action that transforms an input set of flows to a different set of flows. For example, the function F1 in Figure 1.1 inputs the electrical energy flow EE1 and outputs the mechanical work flow MW<sub>1</sub>. The function F2 inputs this flow and outputs two other mechanical work flows, MW<sub>2</sub> and MW<sub>3</sub>. A function is a description of an action, not of the device that performs that action [1, 2]. For example, the function F1 could be mapped to any device that matches this action description, such as an electric motor or a solenoid.

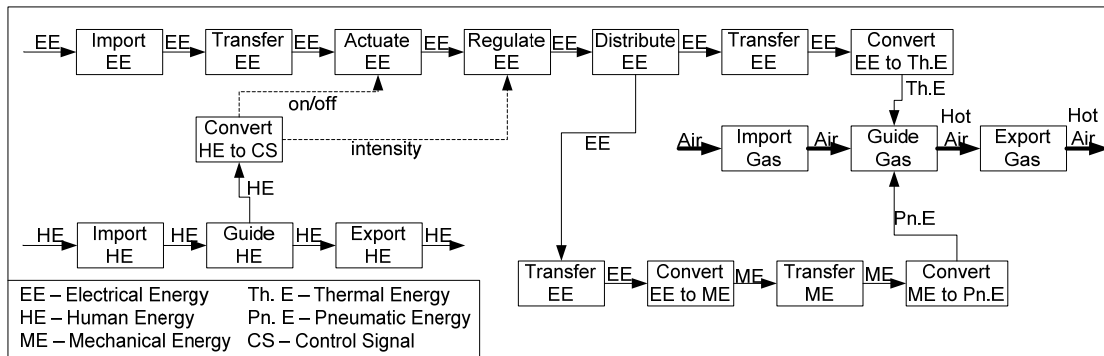
For the purpose of formalizing, it is important to distinguish specific function instances from the generic class named Verb, from which those instances are derived. Function Verb or Verb (proper noun, capitalized) is the class from which flow instance are derived and subclasses of more specific a flow types are inherited to describe templates or types of action, along with the count and types of flows acceptable as inputs and outputs to those actions. The term “a verb” or “verbs” refer to one or more of these classes of actions, while “a function” is an instance of one of the verbs. For example, the function F1 in Figure 1.1 is an instance of the verb Convert in the Functional Basis vocabulary, while the function F2 is an instance of Distribute [26].

If the input and output set of flows are identical, that is, if the count, types, and all attribute values defining flow states are the same between input and output, the function is void, since no transformative action is required to change a flow-set to itself. For a function to be valid, each incoming flow to the function must be different from at least one of its derivatives at the output side. In the graph-based function structure representation, functions are shown as labeled blocks, as shown in Figure 1.2.

### **1.2.8 Function Structure**

Function Structure (proper noun, capitalized) is a representation for describing the functionality of an artifact as graphs [1, 2, 49]. Figure 1.2 shows a model based on this representation. These models are called function structures (common noun, small letters). This model describes a commercial hairdryer product and is available in the

Design Repository, which is a web-based archive of design information of electromechanical products [50-53].



**Figure 1.2: Function structure model of a hairdryer stored in the Design Repository<sup>4</sup>**

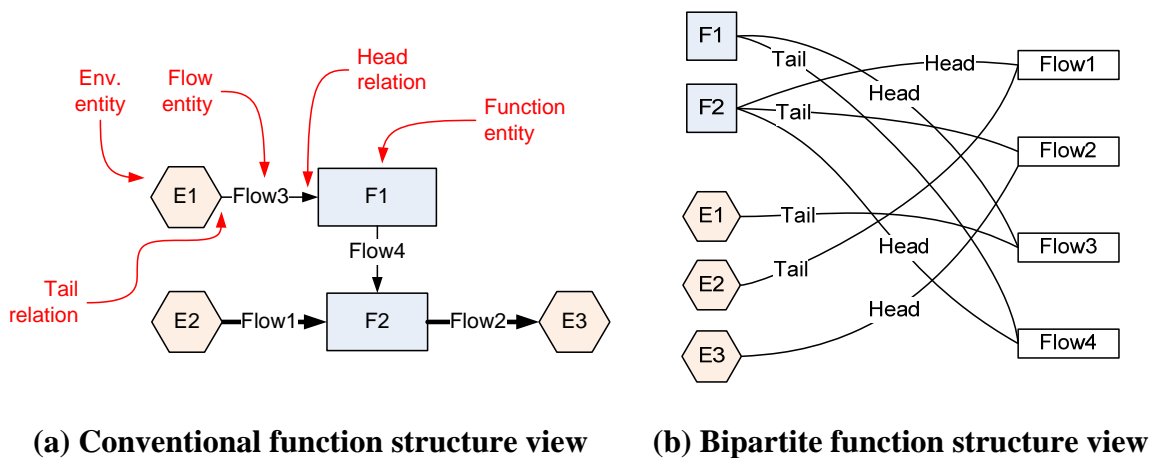
By the conventional definition of the Function Structure representation [1, 2], the edges are flows (Noun instances) and the vertices are functions (Verb instances). In graph theory [49], a vertex conventionally means an entity and an edge is a relation between two entities. Thus, in function structures, flows are treated as relations between functions. However, since every edge has two ends, a relation implies two entities that are related. Then, the model in Figure 1.2 violates the standard construct of graphs, as it contains flows that are connected to only one function. The free ends of these flows imply input or output of flows to the modeled system from the environment, which is

<sup>4</sup> <http://repository.designengineeringlab.org>, accessed on June 11, 2011

denoted by the large rectangle, instead of additional vertices. Thus, while the Function Structure representation is based on graphs, it uses modified graph constructs.

In the proposed representation, functions, flows, and environment instances are all treated as entities, while their connections are relations, as labeled in Figure 1.3a. The straight arrows in this figure are flows, while the curved arrows are label leaders. When the flows are drawn as vertices instead of edges, the function structure model becomes a bipartite graph with two partitions of vertices and two relation types between the partitions. The two types of entities, shown in the left and right partitions of Figure 1.3b, are (1) the nodes of the function structure model (functions and environments) and (2) the flows. The two types of relations, shown by the edges of this graph, are Tail and Head, indicating if a node (function or environment) is the tail or head node of a flow. Figure 1.3a and Figure 1.3b are two isomorphic views of the same model and show the same topologic connections between functions and flows. For example, the two relations attached to Flow1 in Figure 1.3b indicate that Flow1 has its tail attached to E2 and its head attached to F2, which can be verified from Figure 1.3a.





**Figure 1.3: Conventional and bipartite views of a function structure model**

While “vertex” and “edge” are used to describe graphs in general, the words “node” and “flow” are used in this dissertation to describe function structure models. Nodes include functions and environments, since in the conventional view of function structures these two entities form the vertices of the graph. The flows are “edges” (relations) in the conventional view, but “vertices” (entities) in the bipartite view. In all types of graphs, “vertices” mean entities and “edges” mean relations. This convention is carried throughout this document.

### 1.2.9 Topology

Topology of a function structure graph is the arrangement of connection between the functions and the flows. In order to uniquely identify a function structure, its functions, flows, and its topology must be identified. The same set of functions and flows can be connected in different arrangements to produce different function structures that differ only in terms of topology. In Figure 1.3b, the exact set of connections—

including the type of connections and the connected entities to each connection—defines the topology of the bipartite graph.

Next, high-level requirements for the early design representation proposed in this dissertation are mentioned. These requirements are analyzed in greater detail in the next chapter and a more rigorous treatment of identifying specific reasoning needs is presented in Chapter 4.

### **1.3 Summary of High-Level Requirements for the Representation**

In order to support analytical reasoning in the early design, as outlined in Section 1.1, a candidate representation must satisfy a set of high-level requirements, as summarized here. The resulting representation is checked against this requirement list at the end of the dissertation, in Chapter 10.

1. **Coverage over multiple physics domains:** A wide range of physical and mechanical engineering principles and devices should be possible to model and used in early reasoning. Specifically, basic phenomena of electrical, mechanical, and thermal energy forms and their interaction with various material forms should be describable, since the principles necessary to solve novel, open-ended design problems are difficult to foresee.
2. **Domain-independence of physics laws:** The representation must formalize mechanical functions using physics laws that are generally applicable to all domains of mechanical design, rather than incrementally adding specific knowledge and design rules from different domains. Specifically, the

principles of (1) conservation and (2) irreversibility must be included in the representation.

3. **Physics-based concreteness:** The entities, relations, attributes, and grammar rules of the representation should support constructing models that are consistent with the principles of conservation and irreversibility. The representation should also support analyzing models through algorithmic reasoning against these two principles of physics.
4. **Normative and descriptive modeling support:** The representation must support descriptive modeling of existing design concepts, devices, or physical principles. The representation should also support normative modeling of new design concepts.
5. **Qualitative modeling and reasoning support:** The representation must allow the designer to describe a design even when quantitative information is not available. It must allow drawing qualitative inferences of two types from the models based on (1) conservation and (2) irreversibility.
6. **Extendibility:** The representation must support extendibility of the following types.
  - a. **Quantitative reasoning extension:** In the future, the representation should be able to describe quantitative details of a model and support reasoning using that additional quantitative information.
  - b. **Causal reasoning extension:** In the future, the representation should be able to describe causal relations between functions and flows, in

order to support physics-based causal description and predictive analysis of early design.

7. **Scalability:** The representation should support modeling and reasoning on function structure graphs that vary in number of nodes and edges.
8. **Consistency:** It should be impossible to infer two statements P and Q through logical deductions from the declarations made within the representation—such as the definitions of verbs, nouns, relations, constraints, and attribute definitions—such that  $P = \neg Q$ .
9. **Validity:** The representation must be valid against the principles of conservation and irreversibility. Specifically, if a model implies a violation of any of these principles, the representation should support a reasoning algorithm that can detect that violation.

In order to build a representation that satisfies these requirements, two research questions must be answered, which are discussed next.

#### **1.4 Research Questions, Hypotheses, and Tasks**

This section summarizes the research questions answered in this dissertation, the hypotheses of this research, and the tasks used to test the hypotheses and answer the questions. There are two highest-level questions in this research. Three research hypotheses are identified against these two main questions. Each main question is answered through multiple tasks, each of which answers a sub-question under the main question. Table 1.1 and Table 1.2 present these two research questions, and their

hypotheses, sub-questions, and tasks. The questions are numbered as “RQ”, the hypotheses are numbered as “RH”, and the tasks are numbered corresponding to each sub-question, as “Task”. The section numbers in the document that present each task are also mentioned.

**Table 1.1: Research Question 1, hypotheses, and tasks**

Main research question	<p><b>RQ-1.</b> What are the entities, relations, attributes, and grammar rules necessary to formalize the Function Structure representation, in order to support (1) consistent models and (2) analytical computational reasoning on concepts based on conservation and irreversibility?</p>
Hypotheses	<p><b>RH-1.</b> The entities, relations, and attributes shown in Figure 5.1 and the grammar rules of Section 5.2 can support consistent modeling and conservation-based reasoning on concepts.</p> <p><b>RH-2.</b> The representation shown in Figure 7.2, including the grammar rules of Section 5.2, can support irreversibility-based reasoning on concepts.</p>
Sub-questions and tasks	<p><b>RQ-1.1.</b> What specific physics-based analytical tasks should be supported?</p> <p><b>Task 1. Reasoning Discovery:</b> Systematic discovery of reasoning needs through a modeling exercise</p>

	(Section 4.1).
<b>RQ-1.2.</b>	Are these reasoning tasks algorithmically solvable?  <b>Task 2. Algorithmic Deduction:</b> Algorithms for performing the reasoning tasks from the previous task (Section 4.2)
<b>RQ-1.3.</b>	What information elements must be captured to support the algorithms?  <b>Task 3. Information Extraction:</b> Analysis of the algorithms to identify individual data elements (Section 4.3)
<b>RQ-1.4.</b>	Is the representation internally consistent?  <b>Task 4. Consistency Verification:</b> Logical inspection and ontological consistency checking through Protégé OWL (Section 6.1)
<b>RQ-1.5.</b>	Can the representation support physics-based reasoning in early design?  <b>Task 5. Validation of Conservation:</b> Development of a software tool to demonstrate modeling and conservation-based reasoning (Section 6.2).  <b>Task 6. Validation of Irreversibility:</b> Extension of the software tool to demonstrate irreversibility-based reasoning (Chapter 7).

Table 1.2 explains the second research question, its hypotheses, sub-questions, and tasks.

**Table 1.2: Research Question 2, hypotheses, and tasks**

Main research question	<b>RQ-2.</b> At the physics-based concreteness level, is there a finite set of verbs that can describe a variety of physical phenomena and mechanical engineering principles as functions?
Hypotheses	<b>RH-3.</b> The eleven verbs presented in Chapter 8 (Table 8.14) can describe principles from physics and mechanical engineering involving electrical, mechanical, and thermal energy.
Sub-questions and tasks	<b>RQ-2.1.</b> Does the proposed verb set provide modeling coverage over a variety of physics and mechanical engineering principles and devices?  <b>Task 7. Modeling Coverage Testing:</b> Description of principles of physics and mechanical engineering through function structure models involving electrical, mechanical, and thermal energy and material forms (Sections 9.1, 9.2).
	<b>RQ-2.2.</b> Can it support consistent descriptive modeling of existing devices?  <b>Task 8. Descriptive Modeling:</b> Reconstruction of two

	models from the Design Repository (Section 9.4).
	<p><b>RQ-2.3.</b> Can it support consistent normative modeling of new design concepts?</p> <p><b>Task 9. Normative Modeling:</b> Modeling of one new design concept using the proposed vocabulary (Section 9.5).</p>

The answer to these research questions and hypotheses are presented in the concluding chapter (Chapter 10). Next, a brief overview of the solution, the formal representation of mechanical functions developed in this research, is presented.

## 1.5 Solution Overview

The proposed representation is presented in three layers. As discussed in Section 1.2.2, a formal representation is defined with its vocabulary of entities, relations, attributes, and local grammar rules. The three layers of the representation are summarized below.

1. **Layer One** formalizes the graph-based Function Structure representation and supports conservation-based reasoning. It comprises of the following information elements.
  - a. **Six Entities:** Function, source, sink, material, energy, and signal (Figure 5.1).
  - b. **Five Relations:** HeadNode, TailNode, CarrierFlow, Child\_M, and Child\_E (Table 5.12)



- c. **Five Attributes:** GivenName, HeadPoint, TailPoint, GeometricCenter, and AnchorPoints (Table 5.13)
  - d. **33 Grammar Rules:** These rules are described in Section 5.2
2. **Layer Two** is an extension of the first layer that supports irreversibility reasoning by including three additional attributes: IsResidual, Power, and Efficiency (Figure 7.2).
3. **Layer Three** further extends the previous two layers and proposes a physics-based vocabulary of verbs for function model construction and computational reasoning (Chapter 8). Three types of verbs are proposed.
- a. **Seven Energy Verbs in a Two-Level Taxonomy:** TypeChange\_E, Transfer\_E, Change\_E, Store\_E, and Supply\_E. Transfer\_E has two sub-verbs: Conduct\_E and Radiate\_E (Section 8.2.1).
  - b. **Two Material Verbs:** Energize\_M and DeEnergize\_M (Section 8.2.3)
  - c. **Two Topologic Verbs:** Logical\_Branch and Logical\_Unite (Section 8.2.4)

Each layer is validated after it is presented. The internal consistency and external validity of Layer One is established in Chapter 6, where the representation is logically examined for exhaustiveness of grammar, ontologically examined for consistency, and implemented in a software tool named Concept Modeler – ConMod to demonstrate its ability to support consistent model construction and its validity against the conservation principle. Layer Two is validated in Chapter 7 by extending ConMod to implement the new attributes and new algorithms that can detect violations of the irreversibility

principle, both qualitatively and quantitatively. Layer Three is validated in Chapter 9, where the new vocabulary of verbs is used to model principles of physics and the mechanical engineering sciences and further applied to describe existing products and new design concepts through function models using the proposed vocabulary.

In summary, this chapter provides a high-level view of the entire research. It introduces the research problem and establishes the overall motivation for solving it, it identifies (pending elaborate analysis) the requirements that a representation must satisfy in order to solve the problem, and it lists the research questions that must be answered in order to design that formal representation. The chapter ends with a brief overview of the solution. In the following section, the previous and contemporary advances in function-based design formalization are discussed.

## CHAPTER 2. REVIEW OF FUNCTION-BASED DESIGN

Historically, research in the representation of mechanical functions has been conducted from two viewpoints. In Artificial Intelligence (AI), multiple models exist mainly to support device description, cause-and-effect explanation, and design synthesis. The second viewpoint, referred to here as the *engineering design view*, uses primarily one representation—the graph-based Function Structure—to support different design reasoning. These views are discussed below.

### 2.1 The AI Views of Function-Based Design

Function-based reasoning about artificial systems has been a topic of early interest in artificial intelligence and cognition research [54-57], mainly due to the character of design that design reasoning must aid the creative process of synthesizing solution to a new problem, as opposed to explaining the workings of an existing device [55, 56]. Consequently, design reasoning must use a representation that captures the designer's intent—a problem that is central in the models within the AI view [58]. To this end, multiple approaches for automating function-based thinking have been explored, including representations [7, 11, 12, 59-63], languages [19, 64], ontologies [29, 65, 66], and their software implementation [16, 67].

This view recognizes that “how a device works” (behavior [8]) is a more of a scientific problem that is less dependent on the observer's viewpoints than “what a device is for” or “what does a device do for human needs” (function [8, 16, 61, 68, 69]). The

description of function cannot be isolated from the design problem, the designer, his intent, and his view of the design problem, which is together described as the situatedness of the design [10, 11]. Consequently, device function is described as the interaction between several of these elements. Gero describes functions as the required actions of a device and proposes the Function-Behavior-Structure model (FBS) [8, 10, 11, 70], using the expected behavior (Be), the Structure of the device (S), the actual behavior of the structure (Bs), and the process of iteration called reformulation, through which a designer attempts to match Bs with Be, while both evolve with the iterations. Alternately, function is defined as “the relation between the goal of a human user and the behavior of a system” [69]. These models have been used to explain design creativity [9] and later included situatedness [11]: the dynamic situation where the information available to and represented in design influences the designer’s decisions.

A similar representation is the Function-Behavior-State model that defines functions as “a description of (the device’s) behavior abstracted by human through recognition of the behavior in order to utilize it” [59, 71]. This model is intended to support problem decomposition and is extended into a design tool named the FBS-modeler [17] that builds relations between sub-functions to structural features, and further to physical states of those structures that enable performing the functions. The FBS-modeler has been experimentally applied in reducing functional redundancy of electromechanical devices [16].

Another model, named Structure-Behavior-Function, defines function as a set of the input and output states of a device and the behavior that causes state transformation [68]. This model allows building design patterns that capture information about the structure, behavior, and function of devices, simulating “the learning of high-level abstractions and their use in reminding and adaptation” in future [72, 73]. This model supports analogical reasoning and is implemented in case-based reasoning tools such as IDeAL [12, 73] and Kritik [68]. Other approaches attempt to synthesize mechanisms using a representation that describes functions as transmissions of forces and motions in input-output format, and then selecting candidates from a pool using topological and spatial selection criteria [74-76].

Similarly, Functional Representation (FR) defines a device's function from two viewpoints: the device's effect on the environment [60, 62], and in terms of an agent's view of the device, called the “device-centric view” [61]. This model supports failure diagnosis through causal analysis [77], and was implemented in a language named Causal Functional Representation Language (CFRL) [13, 19]. CFRL describes function as the triple  $\{D_F, C_F, G_F\}$ , indicating the device, the context of the device's application, and the goal or desire of the user [19]. CFRL can describe how a device works using causal process descriptions [13].

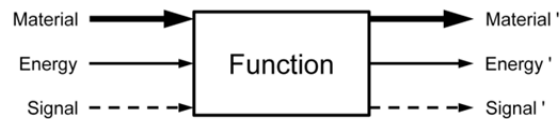
Function and Behavior Representation Language (FBRL) [64] is a representation that captures function and behavior. Behavior is described in terms of “objects” that are input and output through “ports” attached to a device, thus making behavior in FBRL

similar to function in engineering design literature [1, 2, 64]. Function, in FBRL, is a “topping” on the behavior that captures the “goal” of the device, and is described by a four-term vocabulary: ToMake, ToPrevent, ToControl, and ToMaintain [78]. This representation has been shown to support high-level explanation generation for mechanical devices, especially in seven identified categories: function of a component in a system (what a component contributes to the overall function), change of scope (resolution of the observer’s view of the explanation), occurrence of a fault (causal reasoning), use of the ToPrevent function (negation of an action), reason why an output is generated, the reason why an output is not generated (causal), and hypothetical simulations (what-if analysis). Several related ontological classification of functions have been proposed that support design synthesis reasoning of different types [79, 80]. These reasoning types, although identified out of a different need (explanation generation) than the current research motivations (physics-based concept checking), formulate a baseline for concept-level automated reasoning needs.

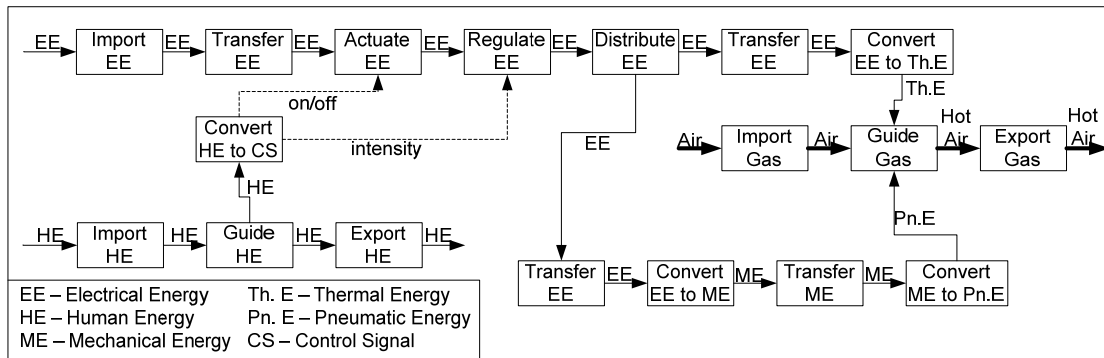
In summary, the AI models of device function are inspired by the complex interaction between multiple entities and are primarily descriptive. These are more holistic views of function and are subject to the difficulties of modeling intentionality of human agents in the design and use of a device [58]. By contrast, the engineering design view, described next, takes a simpler view of functions as transformations of material, energy, and signal and supports some reasoning.

## 2.2 The Engineering Design View of Function-Based Design

This viewpoint defines function as transformative actions between the input and output flows in a system [1, 81-83]. A graph-based representation called the Function Structure is widely studied to describe these transformations [1, 3-6, 21, 25, 84-88], where the nodes are the transformative actions (functions) and the edges are the objects of actions (flows) of three types: material, energy, and signal passing through the device. Figure 2.1 shows the generic graphic template of a function with all three flow types at input and output. The hairdryer model of Figure 1.2 (repeated in Figure 2.2) is a function structure model produced by connecting individual functions performed by a device in a network.



**Figure 2.1: The transformative view of device function [1]**



**Figure 2.2: Function structure of a hairdryer stored in the Design Repository<sup>5</sup>**

Beyond these basic graph-theoretic constructs, the formalization of this representation has been primarily empirical, as opposed to theoretical or logical. To help formalize this representation, controlled vocabularies of functions and flows are proposed, typically through empirical observations, where actions and flows within mechanical devices and systems are observed and cataloged. Examples include the discovery and cataloging of mechanical functions through engineering forensic studies of US Army helicopters by Collins et al. [89] and the function discovery and cataloging of electromechanical consumer products stored in the Design Repository [50, 51, 53]. The discovered functions and flows are typically stored as verbs and nouns in controlled vocabularies for use in future models. Examples include the vocabulary of 46 elemental functions and forty adjectives proposed by Collins et al. [89], the four functions—Motion, Control, Power, and Enclose—proposed by Kirschman and Fadel [90], and the

<sup>5</sup> <http://repository.designengineeringlab.org/> accessed on August 17, 2011



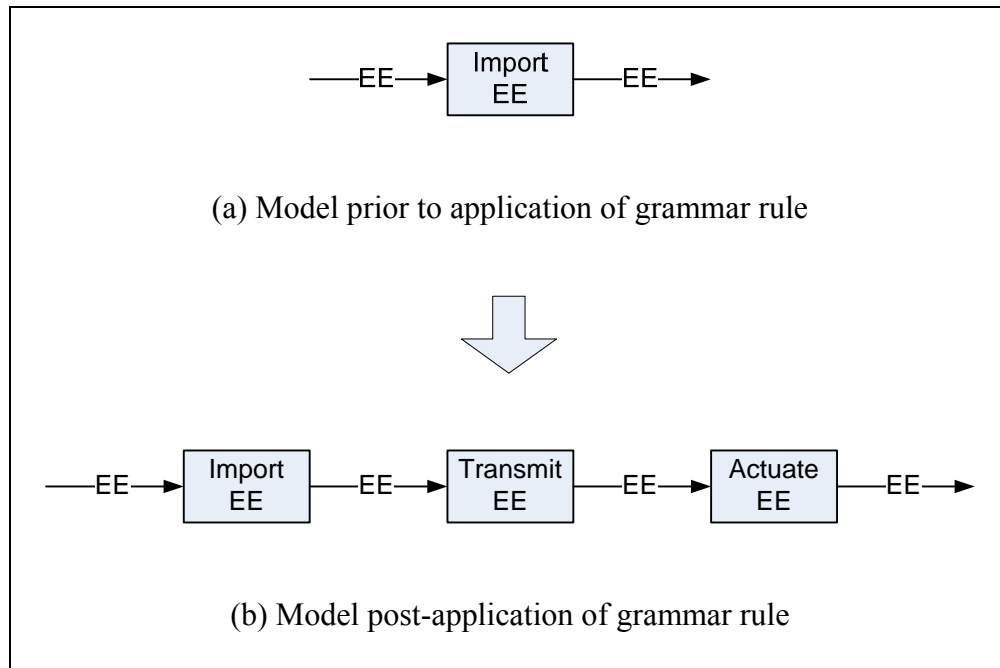
vocabulary of functions in electromechanical products compiled by Szykman et al. [47], and the Functional Basis [25, 26], which is a vocabulary of 53 function verbs and 45 flow nouns organized in a three-level hierarchy.

An anticipated benefit of controlled vocabularies is that they can be used to enforce consistency of term usage in function models. However, for ensuring this consistency, the vocabulary must be consistent itself and model construction must be additionally controlled through grammar rules (modeling guidelines). At present, the formalism for constructing Function Structure models is not developed beyond these vocabularies and grammar rules have not been proposed. This research gap is the one directly addressed in this dissertation research and therefore, this gap is separately discussed in Section 2.5.

Though this second viewpoint is largely based on empirical discovery of functions observed in existing products, it can be used to support forward design projects by providing means to reuse the discovered functional knowledge in new problems. For example, the verbs and nouns of the Functional Basis are identified through reverse engineering of consumer products [2], whose models and other design information are catalogued in the Design Repository [50, 52], as they are generated. Once archived, several design tools are developed that use this function information to generate concepts for new designs [84, 91], to analyze similarity between concepts [88, 92, 93], to analyze or predict failure modes in the conceptual phase [94-97], to decompose functional concepts to smaller problems [5, 6], and to configure component structures [21, 98]. It

must be emphasized that while these tools are computational, they do not perform reasoning on the models directly to draw inferences about their physical behavior or validity against natural laws. Thus, the gap addressed by this dissertation research (Section 2.5) is not addressed by existing computer tools. The justification behind this reuse-based approach is that a large share of design problems is adaptive or variant [99, 100], thus justifying reuse of previous design solutions in new problems.

To support design synthesis automation, graph grammar-based algorithms are proposed that start with a high-level function structure and add or modify details within the model to decompose it in multiple combinatory ways to synthesize multiple solution concepts [5, 6]. The graph-grammar rules [101] are based on historical trends of model topology transformation in the Design Repository models. For example, in the electromechanical products within the database, once electrical energy is imported in the modeled system, it is typically transferred to a switch, where it is actuated by a user. Since the Design Repository contains many models of these products, this trend appears as a generative graph-grammar rule, as illustrated in Figure 2.3. When this rule detects a model construct as shown in Figure 2.3a, it transforms it into Figure 2.3b. Thus, these rules are generative grammar rules [102]. However, these grammar rules are purely trend-based; they do not use the definitions of function verbs or flow nouns to perform reasoning and do not verify that the definition and the use of a term is internally consistent and externally valid. In addition, these rules are executed by the computer program and do not address the aforesaid gap of lack of grammar rules for model construction by a human designer.



**Figure 2.3: Sample graph grammar rule for function model synthesis (Redrawn from [6])**

### 2.3 Comparison of the Two Views

Overall, the two directions in function representation research differ in a few ways. The AI representations attempt to include the user's and designer's intent, while the graph-based model does not include that, though in some models in the Design Repository the user's interaction (usage) with the device is captured through flows of human material or human energy [26]. Second, while a function structure captures function as transformations of flows within the system, the AI models traditionally discard this view on the ground that transformation alone is inadequate to capture the entire essence of functions, specifically the user's intent and the artifact's effect on the environment [59]. Third, unlike the function structures, the AI models typically do not

use static vocabularies for terms used in the models. Rather, to support modeling through a free, natural language, several expanding ontologies of functions have been proposed [27-29, 64, 66]. Finally, the AI models have been extended to substantial degree of formalism, where modeling languages such as CFRL [13, 19] and tools such as FBS-modeler [16, 18], IDeAL, and KRITIK [12] have been implemented, at least in academic applications. In the present state of the art, formalization of the function and flow definitions in function structures is limited to the vocabularies of functional terms and their definitions and automated reasoning on the models is not supported yet.

#### **2.4 The Functional Basis and the Design Repository**

The Functional Basis [25, 26] is a vocabulary of 53 function verbs (Table 2.1) and 45 flow nouns (Table 2.2) organized in a three-level hierarchy. This vocabulary was developed at Missouri University of Science & Technology in a joint effort between industry and academia and was later reconciled with a former vocabulary developed at NIST [47]. This vocabulary was incrementally developed by tearing down electromechanical consumer products through the systematic protocol of reverse engineering [2] and cataloging their functional information using function and flow keywords. As new keywords were found necessary to define the products, those keywords (verbs and nouns) were added to the vocabulary to ensure adequate coverage of product variety [25, 26]. Terminally, the collection of terms was found to be adequate to describe the newer products of the same kind and the vocabulary was reconciled to its final, present form.

**Table 2.1: The Functional Basis Verb Set**

<b>Primary</b>	<b>Secondary</b>	<b>Tertiary</b>
Branch	Separate	Divide
		Extract
		Remove
	Distribute	
Channel	Import	
	Export	
	Transfer	Transport
		Transmit
	Guide	Translate
		Rotate
Allow DoF		
Connect	Couple	Join
		Link
	Mix	
Control Magnitude	Actuate	
	Regulate	Increase
		Decrease
	Change	Increment
		Decrement
		Shape
		Condition
Stop	Prevent	
	Inhibit	
Convert	Convert	
Provide	Store	Contain
		Collect
	Supply	Supply
Signal	Sense	Detect
		Measure
	Indicate	Track
		Display
Process		
Support	Stabilize	
	Secure	

**Table 2.2: The Functional Basis Nouns Set**

<b>Primary</b>	<b>Secondary</b>	<b>Tertiary</b>	
Material	Human		
	Gas		
	Liquid		
	Solid		Object
			Particulate
			Composite
	Plasma		
	Mixture		Gas-Gas
			Liquid-Liquid
			Solid-Solid
		Solid-Liquid-Gas	
		Colloidal	
Signal	Status	Auditory	
		Olfactory	
		Tactile	
		Taste	
		Visual	
	Control	Analog	
		Discrete	
Energy	Human		
	Acoustic		
	Biological		
	Chemical		
	Electrical		
	Electromagnetic		Optical
			Solar
	Hydraulic		
	Magnetic		
	Mechanical		Rotational
			Translational
	Pneumatic		
	Radioactive/Nuclear		
Thermal			

The verbs and nouns in the Functional Basis are meant to be used as functions and flows in a function structure. For example, the conversion of electrical energy to mechanical energy in an electric motor can be recorded as a conversion from electrical

energy (EE) to mechanical energy (ME), as shown in Figure 2.4. This specific model uses function verbs and flow nouns from the secondary level of the vocabulary. Further, the hierarchy of terms is used to control the specificity of the terms. For example, to describe the output energy more specifically, the secondary term ME can be replaced with a suitable tertiary term that is a taxonomical child of ME – in this case, rotational mechanical energy (RME). Conversely, if a lower resolution of description is required, the primary term energy (E) can be used, hiding the details that the output energy is rotational (tertiary), or even mechanical (secondary). The provision for switching levels of specificity is claimed to support functional decomposition and ideation support [25].

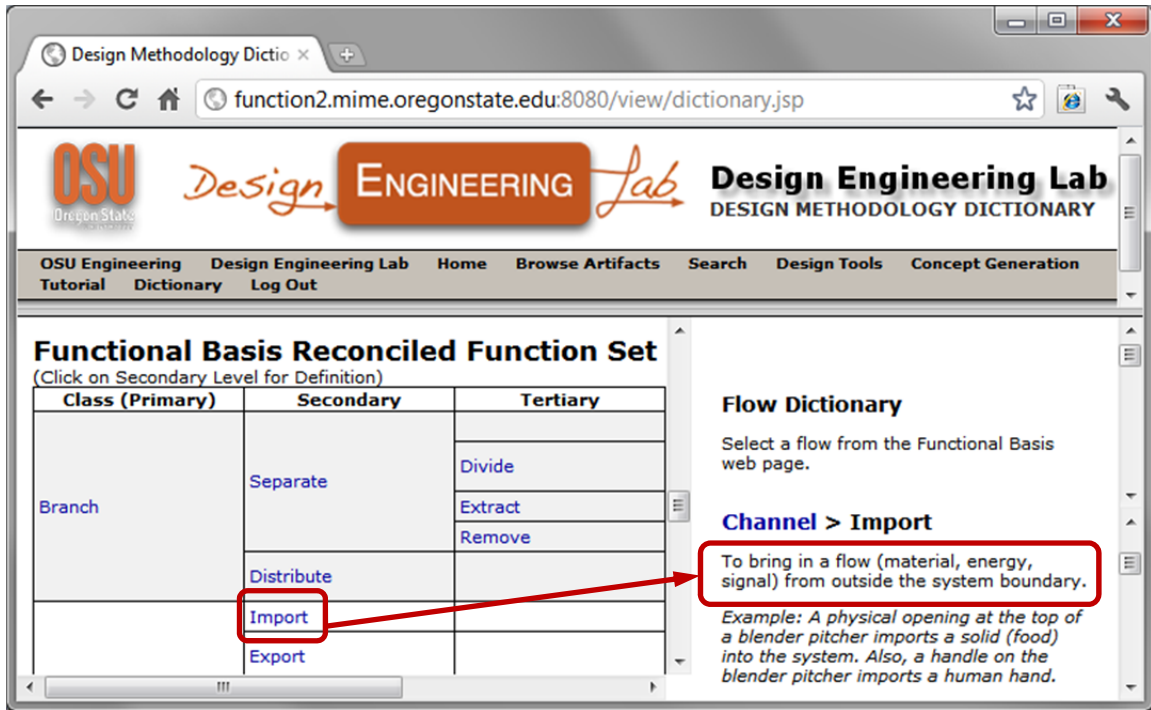


**Figure 2.4: Function structure of an electrical motor using the Functional Basis (secondary level)**

The terms in the vocabulary are defined within Functional Basis literature [26], as well as within the Design Repository webpage<sup>6</sup>, as shown in Figure 2.5. By selecting a term within the vocabulary tables (Import selected in this figure), the definition of the term can be viewed in a different pane.

---

<sup>6</sup> <http://repository.designengineeringlab.org/> accessed on August 17, 2011

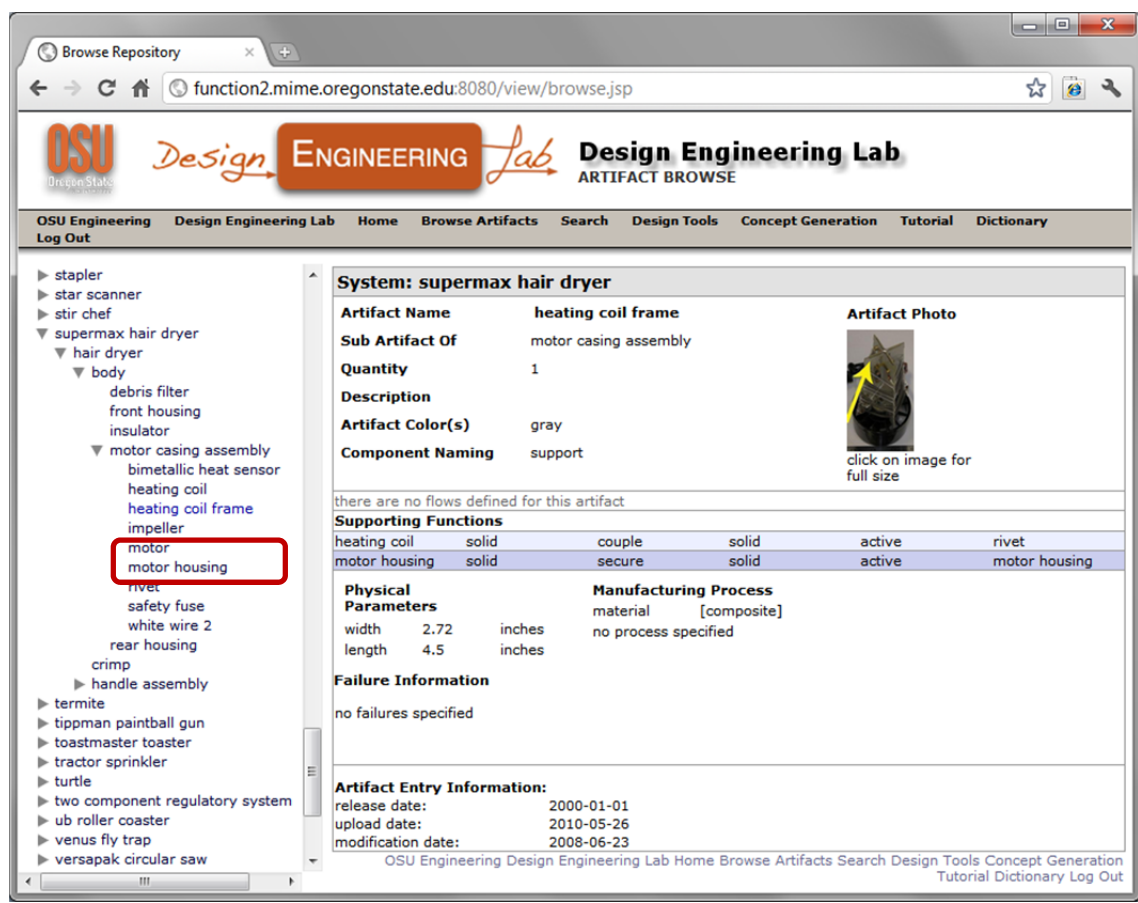


**Figure 2.5: Definitions of terms within the Design Repository**

The Functional Basis has been used to store the reverse engineered design information for approximately 185 electromechanical consumer products produced by the systematic tear-down process described above. This design information is stored in a web-based archive called the Design Repository [50, 52]. In addition to functional information, this repository also stores information about assembly, manufacturing, and physical parameters such as dimensions, mass, and color for the components and subassemblies of the products. Figure 2.6 shows a screenshot of the artifact browser page of this repository, showing one component (heating coil frame) of a specific product (the hairdryer). Notably, since the functional information stored in this archive was created through reverse engineering, where each component was examined in isolation and its functionality was recorded, the functions here are attributed to individual components.



As a result, the functional information stored in this archive is not solution-neutral, which contradicts the recommendations of the design texts that a solution neutral description is preferred for supporting idea generation [1, 3]. As seen from Figure 2.6, the functions are captured in this repository using a list view. Additionally, graph-based function structures for approximately half of the products in the Design Repository are available in drawing form. The model in Figure 2.2 is an example of such a graph-based model. Unfortunately, these drawings are static images and do not support automated reasoning. Additionally, for many products, the list view of functions does not agree with the graph-based description, making the data stored in the archive internally inconsistent.



**Figure 2.6: Artifact browser in the Design Repository showing the heating coil frame of the hairdryer**

The Functional Basis and the Design Repository are widely studied in design research, and have been utilized in constructing several academic design tools and methods [5, 6, 21, 84, 88, 91, 94-96, 98]. For example, the Concept Generator tool suggests component layouts for new design concepts using the component-function matrices of similar products stored in the Design Repository, similar to an automated morphological analysis [84, 91]. Similarly, a failure analysis method, named the Function-Failure Design Method (FFDM), has been proposed to predict potential failure

modes in the conceptual design phase of new designs based on the archived failure history of components performing similar functions [94-96]. This vocabulary has also been used for analyzing functional similarity between products, which relies upon identifying similar occurrences of function-flow pairs between two function structures [88]. The Functional Basis has been extended to formulate a vocabulary of standard mechanical components [21, 98]. Finally, the Functional Basis and Design Repository have been used in creating an automated ideation tool that creates multiple options of decomposing a given function structure based on historical data of functional transitions stored in the Design Repository [5, 6]. These design tools are all based on the premise that the terms in the Functional Basis vocabulary are consistent and adequate. However, this assertion has never been tested through objective empirical studies. In the next section, the Functional Basis vocabulary is critically examined as a benchmark of the state of formalism of vocabulary-based modeling of function structure graphs and the research gap addressed in this dissertation is articulated.

## **2.5 Research Gap Analysis: Lack of Rigor in the Function Structure Formalism**

While function structure models support graph-based visualization, interpretation, and reasoning by human designers, they have not been formalized as a representation. In fact, the only element of a formal representation available is a collection of vocabularies of functions and flows that are formalized to limited extents [25, 26, 47, 103]. Beyond some basic modeling guidelines [87, 104], no formal grammar rule for function structure construction have been proposed. The vocabularies are intended for providing high-level notional descriptions of mechanical actions suitable for modeling of products and

interpreting those models by human designers. These actions are not described with physics-based rigor and contain at least three levels of inconsistency, as explained next. The most commonly used vocabulary of function terms, the Functional Basis, is taken in this research as a benchmark of existing formalism. The lack of rigor in this vocabulary is illustrated here to highlight the types of issues that must be addressed when developing a formal representation of functions. The intent of this exercise is not to point out weaknesses of previous research. As recognized earlier, the Functional Basis is the first consolidated effort toward a finite set of function terms and is widely accepted and used in design research [5, 20, 88, 94, 95]. Its main intent is to support human-driven construction, interpretation, and understanding of function models and to that end, the vocabulary is considered successful by many researchers [5, 6, 20, 84, 88, 91, 94, 95, 101, 105]. The analysis is presented only to identify opportunities of formalization of function definitions for the current dissertation research and to clearly establish the gap this research addresses.

### **2.5.1 Vocabulary-Level Discrepancies**

Vocabulary-level discrepancies are instances where the inclusion of a term in the vocabulary itself causes inconsistency within the vocabulary. For illustration, the energy flow vocabulary of the Functional Basis is repeated in Table 2.3. In this vocabulary, the basis of classification for all terms is not uniform. The nouns are meant to define different **forms** of energy used in mechanical devices [26]. However, while some of these terms such as mechanical, electrical, thermal, or nuclear energy are forms of energy in a physics-based sense, terms such as human energy, biological energy, and solar

energy are classes based on the **source** of energy, rather than its **physical form**. Similarly, hydraulic and pneumatic energies are classes based on the **carrier material medium** of energy, as hydraulic energy is essentially mechanical energy carried by a liquid and pneumatic energy is the same carried by a gas. Mixing multiple bases of classification within the same vocabulary makes the vocabulary non-normal and redundant, and therefore unsuitable for use in formal reasoning. For example, human energy can be mechanical energy, when the energy is available as muscle work that moves an object against a force across a distance. Biological energy obtained from burning wood or coal is a mixture of many physical forms such as thermal and optical. Optical energy and radiated thermal energy are types of electromagnetic radiation, yet these are listed as different types in the vocabulary. Finally, since the material vocabulary contains liquid and gas (Table 2.2) and the energy vocabulary has mechanical energy, the terms hydraulic and pneumatic energy are redundant.

**Table 2.3: Energy flow types within the Functional Basis**

Energy	Human	
	Acoustic	
	Biological	
	Chemical	
	Electrical	
	Electromagnetic	Optical
		Solar
	Hydraulic	
	Magnetic	
	Mechanical	Rotational
		Translational
	Pneumatic	
	Radioactive/Nuclear	
	Thermal	

**2.5.2 Definition-Level Lack of Rigor**

While the existence of certain terms in the vocabulary is shown above to be inconsistent, once those terms are accepted in the vocabulary, their definitions can be further shown to contain lack of rigor. This definition-level lack of rigor happens when

1. A definition contains unexplained and ambiguous terms (ambiguity), or
2. Definitions of two terms are conflicting or redundant mutually (inconsistency)

These two types of definition-level limitations are illustrated below. The conflicts of the definition with natural laws or other body of knowledge external to the definitions is not examined here, since the focus here is to examine only internal inconsistencies, rather than external invalidity.

Ambiguity in Functional Basis Verb Definitions

Table 2.4 shows the notional definition of seven randomly sampled verbs from the Functional Basis. For each verb, keywords that are necessary for consistent interpretation of the definition but are unclear from the definition are identified.

**Table 2.4: Sample verbs and notional definitions from the Functional Basis [26]**

<b>Verb</b>	<b>Notional Definition</b>	<b>Ambiguous Concepts</b>
Extract	To draw, or forcibly pull out, a flow	Draw, force, pull out
Allow DoF	To control the movement of a flow by a force external to the device into one or more directions	Control, movement, force, directions
Inhibit	To significantly restrain a flow, though a portion of the flow continues to be transferred.	Significantly, restrain, portion, transfer
Distribute	To cause a flow (material, energy, signal) to break up. The individual bits are similar to each other and the undistributed flow	Break up, bits, similar
Actuate	To commence the flow of energy, signal, or	Commence

<b>Verb</b>	<b>Notional Definition</b>	<b>Ambiguous Concepts</b>
	material in response to an imported control signal	
Collect	To bring a flow together into one place	Together, place
Contain	To keep a flow within limits	Keep, limits

A major observation from these definitions is their inherent lack of rigor and objectivity. For example, one can ask for the verb Extract “what action constitutes drawing?”, “are there limits to the force that qualifies the action as Extract?”, “what is a definition of pulling out?”, or “from where is the flow pulled out?”. Similarly, for Inhibit, one can ask “how much portion is significant?”, “Since the definition mentions that the unrestrained portion is transferred, is this verb a special case of Transfer? Then, why is it classified under Stop?”, or “how is a portion of a flow defined?”. Similarly, for the verb Contain, one can ask “is the limit mentioned in the definition a limit of space, time, or some other quantity? For example, if a function within a lemonade-making machine keeps the density or sweetness of the liquid within an upper and a lower limit by varying the amount of sugar, should that function be modeled as Contain? Why or why not?” Each of these questions reveals an opportunity where more objective assignment of words could make the definition more rigorous.

#### Internal Inconsistency in Functional Basis Verb Definitions

Some instances of internal inconsistency are easier to detect from direct comparison of the definitions. For instance, while a verb named Separate is present in



the Functional Basis, another verb—Divide—is defined as “To *separate* a flow” [26], thus making one of the two verbs redundant. Some other inconsistencies need a rigorous analysis to be revealed. To this end, Table 2.5 shows syntactic translations of the existing textual definitions of four verbs—Import, Export, Transfer, and Guide—within the Functional Basis, using first order logic-based syntactic statements. To eliminate any negative bias against the level of formalism of these definitions, each definition is treated as a formal statement of logic and is syntactically translated. For example, the textual definition of Import in Functional Basis is “to bring in a flow (material, energy, signal) from outside the system boundary (to inside the system)”. The last clause in parentheses is not a part of the original definition, but is assumed to be true based on applications of this verb in Design Repository models. Since only one flow is mentioned, there is only one element in the both input list and the output list of the formal definition. Further, since the flow is outside the system boundary before importing and belongs within the system after being imported, the functional location (FLoc) of the input and output flows of this verb are assigned as the environment and the system, respectively. In these definitions, the functional location and geometric location are treated to be different concepts. A function model includes a region within a functional space (F\_Space) and describes entities such as functions and flows within that space, in analogy with a geometric solid in the boundary representation [32, 33] that includes a collection of points in the geometric space (G\_Space). Locations within the F\_Space are FLoc, while locations within the G\_Space are GLoc, such as a space occupied by a valid manifold solid [34]. The definition of Transfer illustrates the need for this distinction, as the

function causes a change of GLoc of the flow, as opposed to Import, which changes the flow's FLoc.

**Table 2.5: First order logic-based translation of four Functional Basis verbs**

<pre> Class Import : Function {   In_List = {I<sub>1</sub>};   Out_List = {O<sub>1</sub>};   I<sub>1</sub>.FLoc = FEnv;   O<sub>1</sub>.FLoc = FSystem; } </pre> <p style="text-align: center;"><b>Import</b></p>	<pre> Class Export : Function {   In_List = {I<sub>1</sub>};   Out_List = {O<sub>1</sub>};   I<sub>1</sub>.FLoc = FSystem;   O<sub>1</sub>.FLoc = FEnv; } </pre> <p style="text-align: center;"><b>Export</b></p>
<pre> Class Transfer : Function {   In_List = {I<sub>1</sub>};   Out_List = {O<sub>1</sub>};   I<sub>1</sub>.GLoc ≠ O<sub>1</sub>.GLoc; } </pre> <p style="text-align: center;"><b>Transfer</b></p>	<pre> Class Guide : Function {   In_List = {I<sub>1</sub>};   Out_List = {O<sub>1</sub>};   GPoint Path1 [int n];   I<sub>1</sub>.Course = O<sub>1</sub>.Course = Path1; } </pre> <p style="text-align: center;"><b>Guide</b></p>

Under this formalism, the syntactic definitions should allow identifying the definition of the superclass, Channel, by taking intersection of the four individual classes. A logical definition of Channel is derived from this intersection (Figure 2.7). A comparison of this definition with the textual definition of Channel—“To cause a flow (material, energy, signal) to move from one location to another location”—proves that the

textual definition of Channel does not warrant the verb to be used as a super-class of its subclass verbs. In fact, this textual definition is identical to that of Transfer, which is an anomaly that invalidates the class hierarchical organization of these verbs. Thus, although the Functional Basis vocabulary is hierarchically organized, the terms are not hierarchically valid, when formally inspected.

```
Class Channel : Function
{
  In_List = {I1};
  Out_List = {O1};
}
```

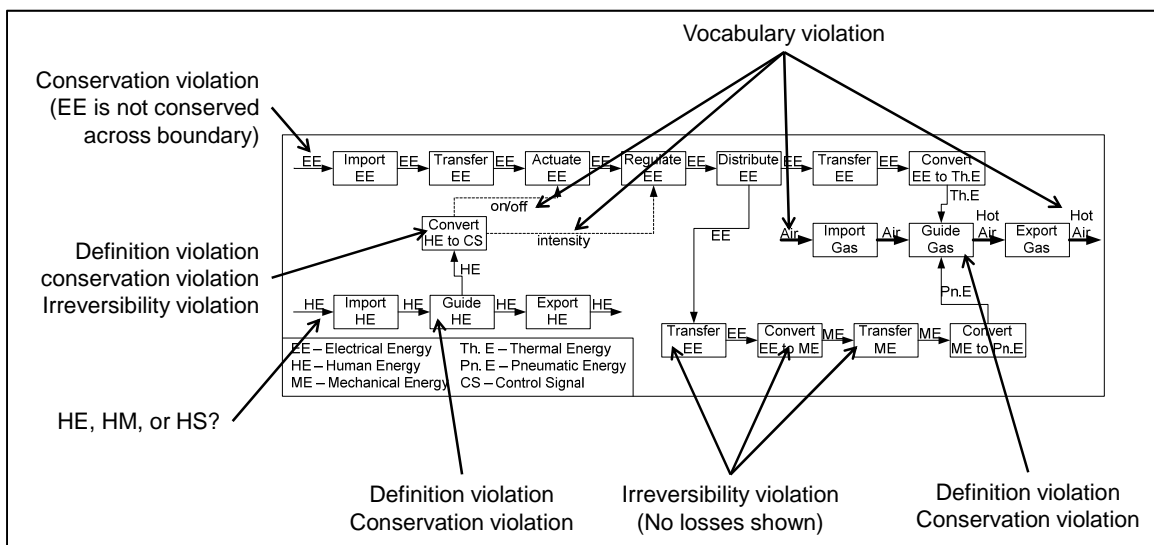
**Figure 2.7: Logical Definition of Channel derived by taking intersection of the definitions in Table 2.5**

Due to these vocabulary-level and definition-level inconsistencies, the vocabulary does not guarantee consistent models, even if it was implemented in a computer tool. Further, the lack of modeling grammar compatible with this vocabulary is another source of lack of formalism, leading to model-level inconsistencies and invalidities, as illustrated next.

### **2.5.3 Model-Level Discrepancies**

Model-level discrepancies are instances where the use of the verbs in a function model creates conflicts mutually, with the definition of its class, or with external natural laws. These instances are not indicative of the inherent definitions of the verbs. Rather, they illustrate the infiltration of errors in models due to the lack of rigorous modeling

grammar rules. Besides some preliminary model construction guidelines [87, 104], modeling rules with Functional Basis verbs and nouns have not been formalized. Figure 2.8 shows the hairdryer function structure of Figure 2.2, with some inconsistencies labeled. Four types of violation: (1) vocabulary violation, (2) definition violation, (3) conservation violation, and (4) irreversibility violation are highlighted. The four instances of vocabulary violation are terms in the model that are not available in the Functional Basis vocabulary. The function “Convert HE to CS” violates conservation from a topological sense, since it inputs an energy flow but produces none. The function “Guide Gas” violates the definition of Guide in the vocabulary—“ To direct the course of a flow along a specific path” [26]—since the definition does not allow adding energy to material flows but the function accomplishes that action. Finally, none of the functions explicitly show losses or residual energy flows produced by the device, thus making the model invalid against the principle of irreversibility.



**Figure 2.8: Illustration of model-level inconsistencies**

In summary, this examination of the most popular vocabulary of function structure modeling—the Functional Basis—identifies lack of logical rigor in its definition and application at different levels. This illustration consolidates the lack of rigor in the current state of the art in function structure modeling. Due to this overall lack of rigor, the models are not suitable for formal analytical reasoning that examines a model’s validity against physics laws or draws inferences in order to discover or predict behavior of the modeled reality (concept). This illustration highlights the need for a formal representation of mechanical functions that is both internally consistent (no self-contradiction or redundancy) and externally valid against the natural laws, specifically, the principles of conservation and irreversibility. The representation proposed in this dissertation research satisfies these criteria, as summarized in the last chapter.

In this context, the need to support conservation-based validity in function models is previously established in the context of Multi-level Flow Modeling (MFM) [106]. Six atomic functions acting on mass and energy (source, sink, storage, balance, transport, and barrier) and four device actions (maintain, produce, destroy, and suppress) are presented with the express purpose of traversing between different levels of abstraction of device description, such as abstract goals and physical functions [106]. However, in the context of the function structure representation, which describes functions as transformation of flows, the laws of conservation have not been explicitly captured in the verbs definitions. While the verbs’ definitions need to be formalized in order to support computational reasoning, this research aims at incorporating the conservation laws and irreversibility-based constructs in those formalized definitions.

## 2.6 Modeling Flexibility and Expressive Power of Notional Terms

While the lack of rigor of the contemporary top-down vocabularies such as the Functional Basis are discussed, it can be argued and observed in the Design Repository models that this lack of rigor indirectly provides a benefit to the modeler, as the definitions do not constrain the modeler with specific details. The lack of rigor allows overloading the meaning of the terms to suit the modeling need at hand, and the terms are perceived as more expressive and flexible for function modeling, at least by human modelers. For example, while the definition of Actuate (Table 2.4) requires that the control signal used to commence the flow be imported, in the hairdryer function structure of Figure 2.8, the signal flow in Actuate EE is not imported from the environment; it is produced by another function. Possibly, the modeler overloaded the definition to retain the idea that to actuate a flow, a control signal should be used, while he ignored the requirement of importing. This case, however, is an example of ignoring an existing constraint (signal must be imported) within a notional definition.

Another benefit of using notional terms in function models is their ability to transmit complex ideas of actions in once instance. Each notional verb describes an action that can be described with multiple physics-based elementary actions proposed in this research. For example, as apparent from the definition of Inhibit (Table 2.4), a material flow is de-energized off its kinetic energy (DeEnergize\_M), but in a manner (not clear in definition) such that some of the energy is available to a portion of the flow, which could be described as an Energize\_M function. The balance energy is probably a residual or transferred to another function (not clear in definition). Similarly, the

definition of Extract, applied to a Material flow, implies that the material experiences a force. If the “pull out” term implies any change in spatial location, the force must perform work, and thus the material must be energized (Energize\_M), while some energy could be lost. Thus, the notional verbs are more expressive than the physics-based verbs—at least to a human modeler—from two accounts: (1) they are less rigorous and afford overloading to meet the current modeling need and (2) they describe more complex actions that can take multiple physics-based verbs to describe.

The above illustration shows that the notional verbs, specifically those in the Functional Basis, are (1) weakly rigorous, but (2) highly expressive. Both characters result from the use of incompletely or informally defined key concepts that must be syntactically captured to formalize these definitions. This task of formalizing the notional definitions is out of the scope of the current dissertation, although it is an important extension that is already underway [48]. With these definitions and their meaning (semantics) captured in syntactic form, semantic reasoning on these terms could be possible in the future. For example, since the verb “Convert” implies that one form or material or energy is converted into another form, by “knowing” this meaning, a computer could reason that an instance of the verb where both the input and the output flows are of the same type, is invalid. In addition, if the computer “knew” that the amount of input energy and the amount of output energies must be equal, as required by the first law of thermodynamics (energy conservation), it could perform quantitative reasoning to detect or prevent instances of this verb where this condition is violated. By extension, if the system “knew” that energy conversion always involves a loss as

consequence of the second law of thermodynamics, it could detect or prevent this type of model-level errors, which could result from designer oversight or the complexity of the system designed. However, a computer “knows” about a domain of discourse such as physics laws only when the entities, relations, attributes, and constraints necessary for storing, algorithmically manipulating, and displaying the knowledge are formally described in a representation. Representing human-interpretable, notional concepts in computer-implementable and computer-reason-able form in this manner is a central challenge of artificial intelligence [107] and this dissertation research proposes a representation of mechanical functions that supports reasoning on these laws, without using this explicit semantic information.

In summary, this chapter briefly reviews the contemporary advances in function-based design within the artificial intelligence and engineering design viewpoints. Based on the gap analysis presented in this chapter, the next chapter develops the requirements on the formal representation in greater detail. The contributions in previous research discussed in this chapter are revisited and critically analyzed in the context of those requirements in the next chapter.



## CHAPTER 3. REQUIREMENTS ANALYSIS FOR THE REPRESENTATION

The purpose of this chapter is to analyze the high-level requirements the formal representation of functions must satisfy in order to support computational analytical reasoning in early design. The requirements identified are based on essential qualities of formal logic systems discussed in literature [30, 31, 108-114], and identified research gaps in function-based design [48, 115, 116]. Coverage, scalability, consistency, and validity are general requirements for logic systems [109, 111-114, 117-121]. Not every system meets each requirement. For example, in mathematics, it can be shown that any set of axioms of natural number arithmetic that is consistent cannot be also complete, as proven by the incompleteness theorem of Kurt Gödel [122]. These requirements are therefore not constraints for a representation. Rather, these are criteria for evaluating how rigorous the representation is. In addition to these criteria, four separate requirements (nos. 2, 3, 4, and 5 in the list below) are identified specifically for the representation problem addressed in this dissertation, based on gaps in design automation research. Each requirement is followed by a brief discussion of how the gap is addressed. Following are the high-level requirements for the proposed representation.

1. Coverage over multiple physics domains (Section 3.1)
2. Domain-independence of physics laws (Section 3.2)
3. Physics-based concreteness of modeling terms (Section 3.3)
4. Normative and descriptive modeling support (Section 3.4)

5. Qualitative modeling and reasoning support (Section 3.5)
6. Extendibility (Section 3.6)
  - a. Quantitative reasoning extension
  - b. Causal reasoning extension
7. Scalability (Section 3.7)
8. Consistency and validity (Section 3.8)

These requirements are analyzed in detail and articulated for the proposed representation in the following subsections.

### **3.1 Coverage over Multiple Physics Domains**

#### **3.1.1 What is Coverage over Physics Domains?**

Coverage of a representation over physics domains means the set of domains of physics whose principles and engineering applications can be modeled with the representation [44, 119]. For example, if representation-A can be used to model only electrical phenomena and representation-B can be used to model both electrical and acoustic phenomena, representation-B is said to have broader coverage over physics than representation-A. Notably, the ultimate practical usefulness of a representation depends on a combination of several qualities such as coverage, consistency, validity, reasoning accuracy, and efficiency. These concepts are treated as orthogonal to each other in this discussion: a change of coverage of a representation does not necessarily imply a change in the other qualities. Thus, the coverage requirement is discussed here without regard to the other requirements in the forthcoming sections.

### 3.1.2 Coverage of Contemporary Representations

Since the physical principles required to solve open-ended, as-yet unsolved design problems are difficult to foresee, in order to be useful for these problems, the representation must support modeling and reasoning based on a wide range of physical phenomena and principles. It is well recognized in artificial intelligence that the broad coverage of a representation usually comes at a cost of reasoning accuracy and efficiency [30]. At one extreme, expert systems that use domain-specific design knowledge can perform fast and accurate reasoning for a specific problem, but cannot solve a different problem even with less speed or accuracy. Their reasoning accuracy and speed rely upon the evolutionary discovery of domain-specific design rules to be reused in future designs—an opportunity that presumably will not exist in novel design problems. Examples include software for configuring automotive subsystems, airplane subsystems, nuclear power plants, or turbo-machine stages.

At the other end of this spectrum lie the general purpose CAD, CAE, and CFD tools that can model and analyze systems from virtually any engineering domain. For example, the same commercial CAD/CAE tool can be used to model subsystems within nuclear power plants, cars, and turbo-machines with equal precision and analyze their behavior with comparable speed and accuracy. However, this broad coverage over design domains—as visible from the ability to model virtually anything geometrically definable—is realized by building the tools on highly generic representations such as the boundary representation [32-34, 123], which comes at the cost of the designer having to construct models for individual designs analyzed. Opportunity for capturing and reusing

domain knowledge is limited, as these systems are not intended to serve a specific problem domain, although several intermediate layers of customization exist between these two extremes, where reusable design elements such as macros and parametric seed parts are used to adapt the general-purpose CAD tools to specific design problems [124, 125].

### **3.1.3 The Coverage Requirement**

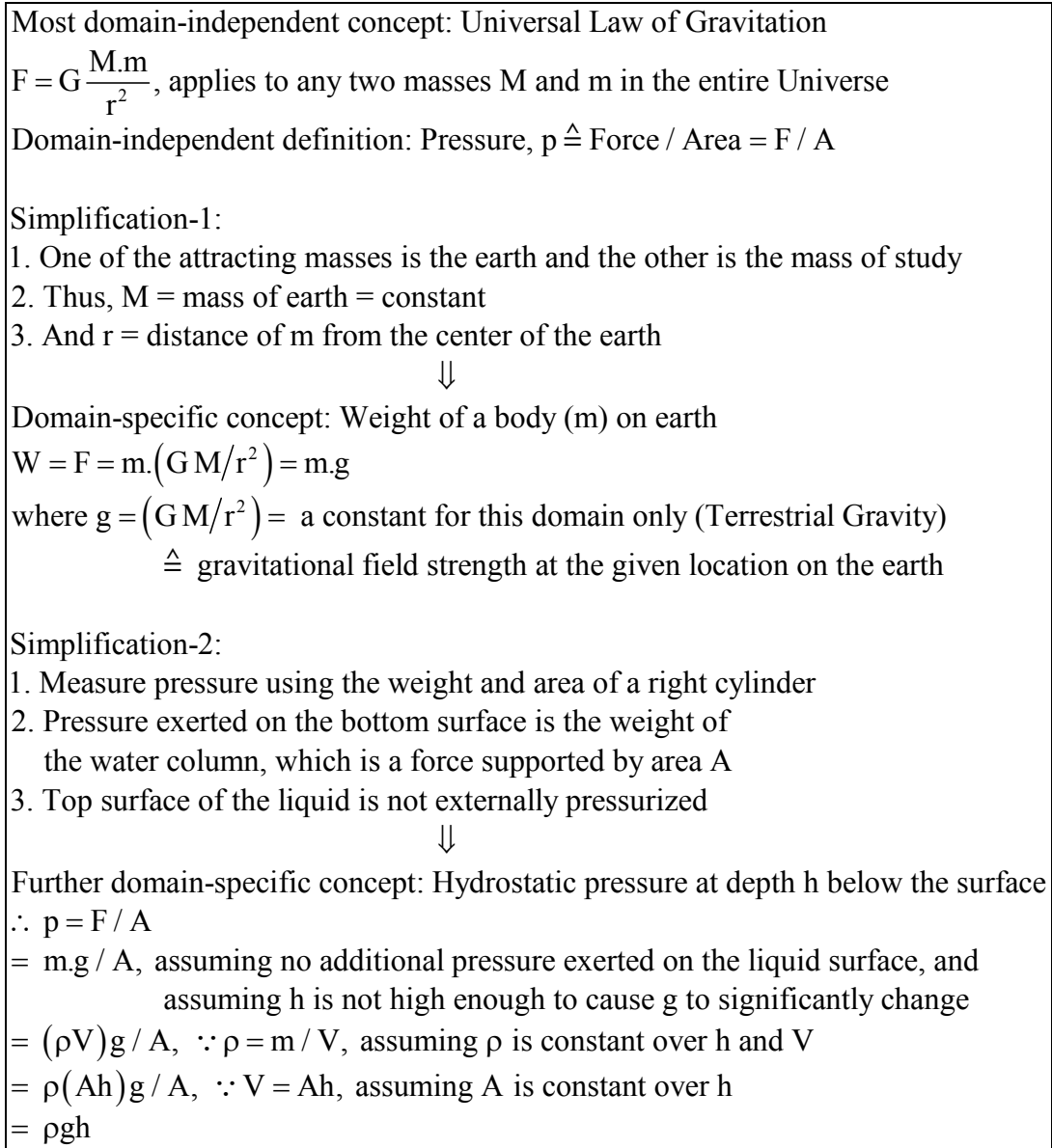
**A wide range of physical and mechanical engineering principles and devices should be possible to model and used in early reasoning. Specifically, basic phenomena of electrical, mechanical, and thermal energy forms and their interaction with various material forms should be describable.** This need is addressed by designing a vocabulary of physics-based atomic functions (Chapter 8) and flows that can model basic physics principles from standard college-level physics and engineering texts [126-129] and could model additional principles in the future. The demonstration of this modeling coverage is presented in Chapter 9.

## **3.2 Domain-Independence of Physics Laws**

### **3.2.1 What is Domain-Dependence?**

Domain-dependence of a physical law describes how restrictive the law is to a given domain of physics, that is, if it can be used to explain phenomena and solve problems only within a certain domain or if it can be applied to a broader set of problems. As physics texts [126-129] commonly illustrate, the laws and equations in a given domain (often analogous to chapters in the book) are derived from more generally

applicable laws through simplifications and restrictive assumptions. By this process, the derived laws, while applicable to those premises under the simplifications, lose their general applicability. For example, hydrostatic pressure under a vertical liquid column of depth  $h$  ( $p = \rho gh$ ) can be explained using the concepts of pressure, volume, density, and weight. While this expression applies to only hydrostatics problems, pressure, volume, density, and weight are more generally applicable concepts. The concept of weight ( $mg$ ) is derived from the Universal Law of Gravitation using the simplification that one of the masses in the equation for universal gravitation is the Earth. Here, the expression  $w = mg$  with  $g = 9.81 \text{ m/s}^2$  is restrictive, as it only represents the force of attraction between a body near the Earth and the Earth, while universal gravitation is a more general concept applicable to any two bodies in the Universe. In fact, gravitation is one of the most fundamental laws of physics that cannot be explained with other classical laws, other than the general theory of relativity [130, 131]. This successive loss of generality caused by adapting the laws and rules (equations) of physics to specific domains such as the Earth (for weight) and hydrostatics (for pressure) is illustrated in Figure 3.1. The figure shows the derivation of the expression for hydrostatic pressure from the Universal Law of Gravitation. Each downward arrow leads to a more domain-specific concept that is derived from a more domain-independent concept, through the simplifications mentioned above the arrow.



**Figure 3.1: Loss of generality of physics equations due to increasing domain-dependence (Adapted from [126])**

While the more general versions of the laws are not inapplicable to the specific domains, the domain-specific derived rules and laws are more practical to use, as they are written in terms of parameters that are easier to measure and control in those domains.

For example, the expression for hydrostatic pressure can be written in terms of universal gravitation (Figure 3.2). This equation could actually be used to cases where the assumptions in  $p = \rho gh$  do not apply, such as for a very tall water column along which  $g$  or  $\rho$  changes. However, for most practical water columns, this equation would not be very useful, since it is expressed in terms of difficult-to-measure parameters such as the mass of the column and the Earth, the distance of the column's center of mass to the earth's, and the universal gravitational constant,  $G$ . Further, it uses the parameter  $A$ , which, for other reasons not captured in Figure 3.2, is insignificant for measuring hydrostatic pressure at a point inside the liquid. The expression  $p = \rho gh$  is not only smaller and simpler; it is expressed in terms of parameters that are much easy to measure and control. However, the  $p = \rho gh$  form only works under the assumptions stated in Figure 3.1 and makes the equation more restrictive.

<p style="text-align: center;">Force of gravitational attraction between the Earth and the liquid column</p> <p style="text-align: center;">Pressure = <math>\frac{\hspace{10em}}{\text{Area of the base of the column}}</math></p> $= \frac{G \frac{M \cdot m}{r^2}}{A} = \frac{G \frac{M_{\text{Earth}} \cdot m_{\text{Water}}}{(r_{\text{Earth-Water}})^2}}{A},$ <p>where <math>r_{\text{Earth-Water}}</math> is the distance between the centers of mass of the Earth and the water column</p>
--

**Figure 3.2: Expression of hydrostatic pressure in terms of universal gravitation**

### 3.2.2 Level of Domain-Dependence of Comparable Representations

The issue of coverage over physics could possibly be addressed in two approaches: (1) by finding the specific physics laws and design rules for different design

domains and incrementally adding them to an extendible representation that will eventually solve a broad array of problems with high accuracy and efficiency, and (2) by finding suitable domain-independent physics laws that apply to a broad array of mechanical systems without being rewritten for specific domains and capture them in a representation. The first approach is taken by multi-domain simulation and solving tools such as Modelica<sup>7</sup> [132-134], which can configure multiple systems within the automotive domain, such as the drive train, air conditioner, and suspension. In academic research, similar approach is used in the 2<sup>nd</sup> CAD tool of conceptual design of electromechanical systems [135], which relies on codifying catalog data of mechanical, fluid, and electrical components. However, while such tools can eventually solve a variety of problems, at any point in time they still remain domain-specific and could not be used to solve a problem whose domain knowledge is not captured yet. Moreover, design rules in these systems are typically represented as algebraic or differential equations and require quantitative specification of design parameters during modeling, which may not be possible in conceptual design, thus rendering this approach unusable for the purpose.

The research in this dissertation takes the second approach. Instead of focusing on domain-specific physical laws and design rules, it captures the laws that are generic enough to apply to any system at any stage of product development, such as the

---

<sup>7</sup> <https://www.modelica.org/>, accessed on August 16, 2011



conservation laws of mass and energy. The degree of domain-dependence of physics laws forms a continuum and choosing the appropriate level is a fundamental challenge in this research. At one end of this spectrum are the most fundamental laws that physicists agree can be used to explain all phenomena studied: (1) conservation laws of mass and energy, (2) conservation of momentum of an isolated system, (3) Newton's law of gravitation, (4) Newton's three laws of motion, (5) Maxwell's laws of electromagnetism, (6) the laws of thermodynamics, and (7) the invariance of speed of light [130, 131, 136]. Arguably, all physical phenomena known to science can be explained using these laws, just as pressure was explained in terms of universal gravitation, albeit often in a much convoluted manner. However, these laws are not directly used to solve problems in physics and certainly not in engineering, because easier and more practical views derived from these fundamental laws exist for different domains. For example, the phenomena of sound propagation in air can be explained by applying the laws of motion to the individual air particles [126] and the concept of pressure of an enclosed gas can be explained from laws of motion and momentum conservation in elastic collision using the kinetic theory of gas [126]. Yet, there exists specific equations for solving problems in sound propagation and pressure [126], thus saving the physicist the labor of deriving every calculation from the fundamental laws. As these physical phenomena are used in engineering design, the laws become further domain-specific, depending on which design parameters are measurable and controlled, and in terms of which parameters the phenomena was discovered and studied. For example, the effect of surging in a gas medium involves both pressure and propagation of the wave front at sonic velocity. Yet,

in centrifugal compressor design—based on the author’s experience of working at a compressor design company—the rules to prevent surging are typically written in terms of empirical laws concerning geometry, gas properties, and flow parameters, rather than in terms of the kinetic theory or the more fundamental laws. Domain-specific modeling and simulation tools such as Modelica typically use design rules at this end of the spectrum.

### 3.2.3 The Domain-Independence Requirement

**The representation must formalize mechanical functions using physics laws that are generally applicable to all domains of mechanical design, rather than incrementally adding specific knowledge and design rules from different domains. Specifically, the principles of conservation and irreversibility must be included in the representation.** To this end, the physics laws in standard college-level physics books [126-129] is deemed appropriate, as engineers are usually familiar with them by education and while these laws are not specific to any class of devices, they could be used to analyze design models at a theoretical level if the models were constructed by composing physical phenomena that these laws govern. For example, if a conceptual model of a clothes ironing machine is constructed in terms of the transfer of mass and energy flows, the representation could support reasoning to determine if the laws of balance are violated in the model or if the efficiency of a subset of processes is within a specified range. This use of college-level, device-independent physics to analyze early design is a key feature of this research and has not been explored previously. Specifically, function structure graphs describe the flow of energy and material: the two

entities that are subject to a fundamental law of physics—the laws of conservation—even at a conceptual level. Thus, these models provide an opportunity to enforce the conservation and balance laws of mass and energy, as realized in the representation.

However, in order to model a device using a function structure that is consistent with physics, the elements of the model—function verbs, flow nouns, their relations, and their modifiers—must be defined with a physics-based concreteness, so that formal reasoning can be supported. This requirement is explained next.

### **3.3 Physics-Based Concreteness of Modeling Terms**

#### **3.3.1 What is Concreteness?**

The adjective “concrete” is used here as an antonym to “abstract”. Traditionally, function verbs are discovered empirically, by observing actions performed by mechanical artifacts and asking: “what could be a verb to describe the observed action for future use in function modeling”? However, when one attempts to describe the actions of a device, multiple descriptions can result depending on the abstractness of the language and the degree of detail sought. For example, when attempting to describe the overall action of a hairdryer, a designer can use any of the following statements. The action in each statement is underlined.

1. “It helps the user to dry hair.”
2. “It dries hair.”
3. “It produces heat for drying hair.”
4. “It produces a stream of hot air.”

5. “It transforms cold air at rest to hot air in motion.”
6. “It converts electrical energy to kinetic and thermal energy and adds them to air.”
7. “It converts electrical energy to heat and kinetic energy of a fan rotor.”
8. (All of the above) + “it converts part of the incoming energy to mechanical, thermal, and acoustic energy, and a magnetic field, and releases them to the surroundings.”

Each of these sentences is a description of the device’s action and each is correct, but they are in increasing order of concreteness and decreasing order of abstractness. The first sentence tells nothing about what the device does, rather, describes what a user could do with it, although it is phrased so that the device is the subject of the sentence. In affordance-based design, this view is called the user-artifact affordance (UAA) of the device [137-140], to illustrate what the device affords, rather than what it “does”, for a user. The subsequent sentences become decreasingly descriptive of the user, his goal (to dry hair), the device’s purpose as perceived by the designer, user, or society (to dry hair), or the device’s surroundings (air). They become more concrete as they focus directly on the device’s **physical actions** and ignore what bigger purpose is served by those actions. The last sentence is a highly concrete description of what the device “does”, rather than what effect it creates, and leaves the description of its function to the spinning of the fan. At this level of concreteness, even the fact that the device draws or delivers air is considered contingent upon the surroundings being filled with air and thus is not a description of the device’s action. Similar hierarchies of abstraction are recognized in

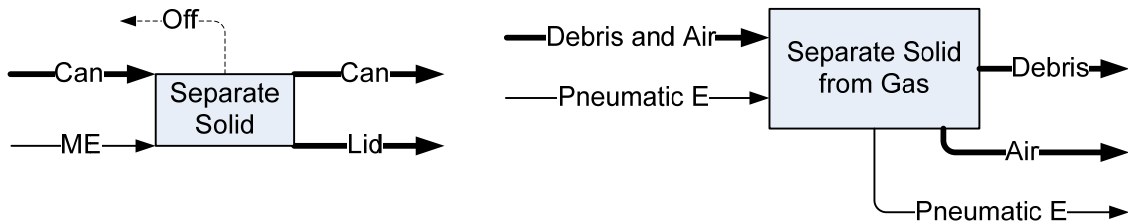
previous research, notably by Morten Lind in his Multilevel Flow Model [106] with three levels: goal, purpose, and function, and by Chandrasekaran who distinguishes between “function as effect” (more abstract) and the “device-centric” view (more concrete) [7, 60-62]. Experimental evidence also supports similar variability in concreteness and level of detail in product description [141, 142].

### **3.3.2 Concreteness of Existing Function Vocabularies**

The reverse engineered function vocabularies, specifically the Functional Basis [26] and Collins’ vocabulary [89], generally define terms at device-centric concreteness. However, these terms are not described directly as physics-based phenomena of the device, possibly because they were not intended to support physics-based reasoning. Rather, they are defined at a level of concreteness a designer is likely to find useful for modeling. For example, the verb Separate is defined in the Functional Basis as “To isolate a flow (material, energy, signal) into distinct components. The separated components are distinct from the flow before separation, as well as each other” [26]. The definition does not describe the physical process that causes separation. Further, function models in the Design Repository<sup>8</sup> reveal that different applications of this verb that are consistent with this definition are realized through different physical phenomena. For example, the use of the verb Separate in the function structures for a can opener and a vacuum cleaner product within this database are shown in Figure 3.3.

---

<sup>8</sup> <http://repository.designengineeringlab.org>, accessed on June 8, 2011



**(a) Use of Separate in the can  
opener model in the Design  
Repository**

**(b) Use of Separate in the shop-vac vacuum  
cleaner model in the Design Repository**

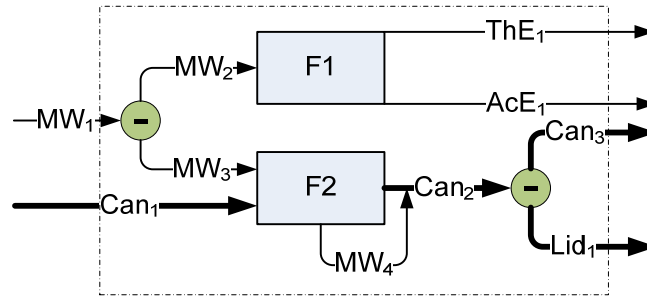
**Figure 3.3: Use of the verb Separate in two different models in the Design  
Repository**

With minor exceptions, the models are topologically similar: both receive a material flow and an energy flow, and output two different material flows derived by separating one from the other. The energy flow is available in some output form, although it is not shown in the can opener model, presumably because of modeler oversight. This oversight is an example of possible model-level inconsistency due to the onus of model consistency being on the designer, instead of being enforced by the representation's consistency, as explained in Section 1.2.4.

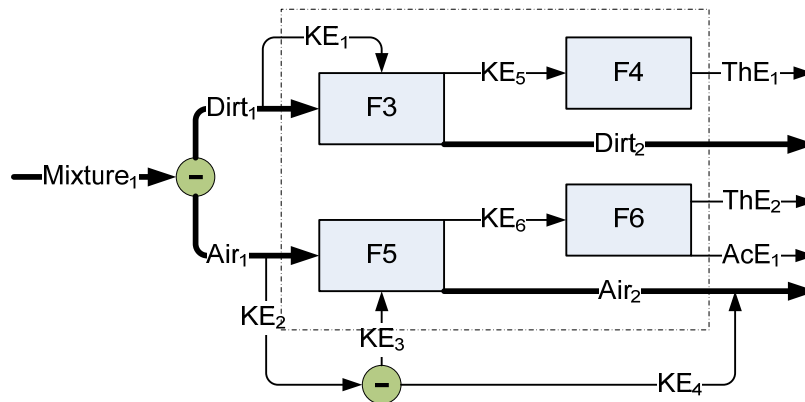
However, the physical principles of these two separation processes are quite different, which is a detail not captured in any of these two models, since the representation used in these models, consisting of the vocabulary [26] and the modeling guidelines [86, 87, 104], do not enforce such detail. For example, separating the lid from a can in a can opener needs mechanical work to be done on the assembly such that the

forces binding the lid to the can are exceeded and the mechanical work is dissipated as heat and sound, while separating dirt from air in a vacuum cleaner filter requires the kinetic energy of the dirt particles to be completely removed without completely removing the kinetic energy of the air, a process that dissipates the absorbed energy as heat and sound. These two processes are graphically shown in Figure 3.4 using a function structure-like modeling construct. The dotted line in each case is the overall instance of Separate, while the blocks inside are decomposition of how material and energy flows interact to cause the separation. The following rules are used: (1) function names are replaced with serial numbers since the identity of these verbs is not important for understanding the models, (2) the circular nodes with a minus sign represent balance between input and output flows for separate accounting of its portions, (3) the types of energy flows entering and leaving the blocks where a material flow is energized (F5) or de-energized (F2, F3) are kept the same without type change, while (4) any type change that happens in these functions is separately shown in other blocks (F1, F4, F6). Additionally, the individual functions or any subset of them is carefully drawn to ensure energy balance and mass balance at least at a qualitative level, by ensuring that for every input material or energy flow, there is at least one output flow of the same type. The symbol MW reads as mechanical work, which replaces the term pneumatic energy or mechanical energy in Figure 3.3. Many of these modeling constructs are result of the design of the representation in this dissertation research, and are clarified later. At present, the purpose of these models is to illustrate that it is possible to construct function

structure-like models to describe functions as physical principles, at least at a qualitative level.



(a) Separation of lid from can



(b) Separation of dirt from air-dirt mixture

**Figure 3.4: Different physics of the same verb in two applications**

The two processes in Figure 3.4 involve different physical phenomena organized in different topologic arrangements, indicating that the definition of Separate does not



map to a unique physical principle. However, this verb is widely used in the Design Repository models: a total of 67 times in 130 models<sup>9</sup>, indicating that the term is found useful by modelers for descriptive function modeling. In general, it can be observed from similar exercises that most of the Functional Basis terms are not defined at physics-based concreteness. The concreteness of these verbs is difficult to pinpoint without a pre-established hierarchy of concreteness. Why this level of concreteness was chosen, whether the level is uniform for all verbs, or whether this concreteness is indeed optimal for descriptive modeling has not been studied objectively. However, this vocabulary is widely used for product modeling in design research and education—more often for descriptive modeling than normative modeling as per published literature—and therefore this concreteness is deemed user-tested for the current research’s purpose. Ultimately, it is accepted on face value that the concreteness of the Functional Basis is at least suitable, if not optimal, for descriptive modeling. This acceptance may lack academic rigor, but it does not influence the outcome of this research and is left for future researchers to reinstate or refute rigorously. For this research’s purpose, the concreteness of the reverse engineered vocabulary terms such as Separate is called **notional** while the level at which the verbs directly describe the physical actions as material and energy balance is called the **physics-based** level of concreteness. The Functional Basis vocabulary is used as a

---

<sup>9</sup> Based on June-2009 data

benchmark of notional terms, since of all the vocabularies, it is the most used in design research and product modeling in design education.

By virtue of this notional abstractness, the Functional Basis verbs are capable of communicating more information in the models than only the transformation of material, energy, and signal, as captured in the correspondent terms in the vocabulary. For example, the verb Extract can be used to describe any action that is notionally described as one of its six correspondents: refine, filter, purify, percolate, strain, and clear [26]. Similarly, the definition of Guide captures that a flow is constrained to a specific path in Euclidean space [26]. Thus, these terms or similar terms at the notional level are highly expressive and are deemed more useful than terms defined with physics-based concreteness for early design modeling and human interpretation of models.

### **3.3.3 The Physics-Based Concreteness Requirement**

While the notional terms can capture more information, this lack of concreteness prevents them from supporting physics-based reasoning. Since notional concepts are difficult to formalize, the Functional Basis terms have never been formally defined for computer implementation and do not support any computational reasoning based directly on their meanings [48]. To address this gap, the verbs in the new representation must be designed to describe physical actions of devices directly, instead of notional actions, so that (1) their definitions can be formalized and (2) physics-based reasoning can be performed based on their definitions. For example, in Figure 3.4a, the ratio between the mechanical work used to cut the lid ( $MW_3$ ) and the total mechanical work consumed by

the model ( $MW_1$ ) could be used in computer reasoning to estimate energy efficiency of the concept. Much of early design reasoning can be done based on the governing physics of mechanical systems. For example, a qualitative understanding of electricity and DC motor principles can be used to reason that an increase in the voltage across the motor terminals would result in higher current through the rotor coils, causing higher torque, and ultimately higher speed of the rotor, assuming constant load. Quantitative knowledge of these principles could additionally support quantitative assessment of this causal chain. It is anticipated that by making the modeling terms (functions, flows) physics-based, many useful reasoning could be supported, while still in the early stage. To this end, **the entities, relations, attributes, and grammar rules of the representation should support constructing models that are consistent with the principles of conservation and irreversibility. The representation should also support analyzing models through algorithmic reasoning against these two principles of physics.** The representation addressed this requirement using a multi-layer vocabulary that evolves from a vocabulary of symbols to two physics-based layers conforming to the first and the second laws of thermodynamics.

### **3.4 Normative and Descriptive Modeling Support**

#### **3.4.1.1 What are Normative and Descriptive Models?**

The distinction between the two model types—normative and descriptive—is discussed in early philosophy and design research [55]. In philosophy and ethics, the word normative means “what should be” and descriptive means “what is” [55]. In

design, these are important concepts, as design begins with a need or the intent to create a solution and ends with an actual solution. Overall, a normative model is one that the modeler considers as an ideal or intended description of an entity and a descriptive model is the actually observed description of an entity.

A normative function structure is a function structure that describes the ideal or intended transformative actions, flows, and their ideal topologic arrangement. It describes functionality that the designer aims to realize.

A descriptive function structure is a function structure that describes the actual transformative actions on flows and their topologic arrangement that occur in an actual device or an existing concept, during one of its modes of use, as observed by the modeler. Since these models describe actual devices, a descriptive function structure is complete when it captures all the functions and flows present in the device.

An important feature of these definitions is that the two types of models are distinguished not based on the “state” of the model as described by their contents, but based on the “process” and “purpose” of constructing them and what the designer “believes” about them. The definition of the normative model does not presuppose correctness, consistency, or feasibility of realizing it in design. Thus, a normative function structure could violate known laws of physics or logic and still be accepted as a normative model. Similarly, the descriptive model definition does not presuppose that the designer was successful in observing and capturing every actual functional detail, as visible in the missing output energy flow from the Separate function in the can opener

model in Figure 3.3. Both normative and descriptive models can be “wrong” or “incomplete” and still be allowed by their definitions. Thus, it is impossible to tell from only observing a model if it is normative or descriptive. However, complete descriptive models are consistent with physics and logic, since real devices obey physics and logic.

### **3.4.2 Characterization of Existing Function Representations**

Most of the existing function representations show support for both normative and descriptive modeling, although some have are designed to serve one better. For example, the Function as Effect model [60] and the Causal Function Representation Language [19] are intended to model existing devices in a descriptive manner and then analyze them using causal reasoning. The Function-Behavior-State model [16, 17, 143] is designed to construct normative models of new design artifacts and analyze their behavior. The Functional Basis [26] and Collins’ vocabulary [89] were created by observing actual device functions in a descriptive manner. However, their terms are used in both descriptive [53] and normative modeling [5, 6, 20, 22]. Most models in the Design Repository are descriptive models, as they are created through systematic reverse engineering of existing products. A graph grammar-based design synthesis tool [5, 6] produces normative options for decomposing a given overall normative function, using trends observed in descriptive models within the Design Repository as reference for its grammar rules. The Function-Behavior-Structure model [8, 10, 11] provides a high-level explanation of how normative models created by the designer evolves through an iterative process called reformulation by comparing the normative model with descriptive models of the available design options.

### 3.4.3 The Normative and Descriptive Modeling Requirement

**The representation must support descriptive modeling of existing design concepts, devices, or physical principles. The representation should also support normative modeling of new design concepts.** It is anticipated that physics-based terms would be more useful for descriptive modeling, as physical actions can be observed directly on the device or concept, thus resulting into physically concrete models. This method of descriptive modeling is different from the method used in existing reverse engineered models, specifically those in the Design Repository constructed using the notional terms of the Functional Basis, in the sense that these models are already physics-based at the time of construction. Reasoning systems that currently use the Design Repository models for building patterns to base their reasoning algorithms, such that the graph-based synthesis tool [6], the failure prediction and diagnosis tools [94-97, 144], and the model similarity detection tool [88] could potentially support more accurate and efficient reasoning, if they were rewritten to use these physics-based models, as these models capture the transformation of material and energy flows in more concrete manner.

For normative modeling, both notional and physics-based terms can be useful, depending on the concreteness of the concept or the state of knowledge of the designer. For this type of models, the representation allows using terms from the notional level vocabulary, or the physics-based vocabulary, or composing new user-defined notional terms from the other vocabulary elements. In each case, the representation supports reasoning to check correctness of the model against the laws of physics captured. It is anticipated that the process of reformulation identified in previous research [8, 10, 11]

can be supported through computer-assisted design through this capability, as the designer can model a normative design without regard to obeying physics, while the tool ensures physics-based correctness of the model at each edit step or on request, thus ensuring that the modeled idea, while still normative, is not in violation of physics or far from realizable.

### 3.5 Qualitative Modeling and Reasoning Support

#### 3.5.1 What is a Qualitative Function Model?

A qualitative function model is a function model that does not contain any quantitative information about the functions and flows. For example model (a) in Figure 3.5 is a qualitative model, while model (b) is quantitative, as it contains quantitative information about the power associated with the mechanical energy flow (ME) and the efficiency of the Convert function.

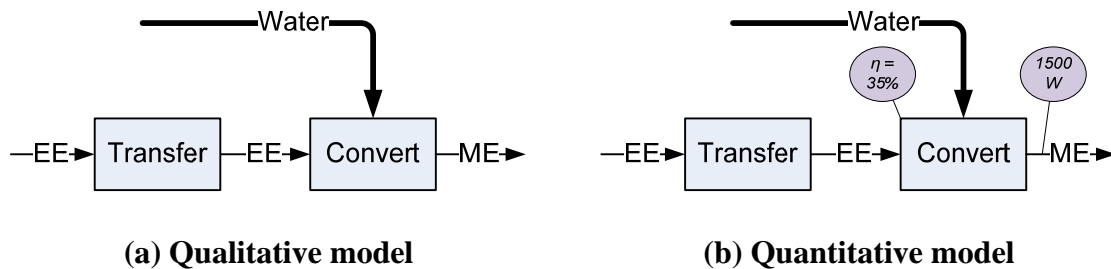


Figure 3.5: A typical function model for energy conversion

#### 3.5.2 What is Physics-Based Qualitative Reasoning?

Qualitative physics is a technique of expressing physics laws and equations in qualitative terms, called confluences [145-147], primarily by capturing how the direction

of change of a parameter in an equation (increase, decrease) causes a corresponding change in other parameters, without using magnitudes for any of the parameters. When applied to a function structure model instead of an equation, an example of **confluence-based** qualitative reasoning is to infer from Figure 3.5(a) that an increase in the amount of energy or power of the input EE flow will cause a corresponding increase in the ME flow. By contrast, quantitative reasoning would be to infer from Figure 3.5(b) that the input EE to the function Convert must be supplied at a rate of 4825.7 Watts, using the power attribute of the output ME flow (1500 W), the efficiency attribute of the function Convert (35%), and the definition of the term efficiency. As seen here, this quantitative reasoning requires quantitative information in the model, and thus the representation must have provisions for holding such information.

Another example of physics-based qualitative reasoning would be to infer that the Convert function violates the **conservation law** of material, as it does not produce any material flow despite receiving one. Similarly, the Convert function in Figure 3.5(a) violates the second law of thermodynamics and **irreversibility** principle, as all of the incoming electrical energy is shown to be converted into mechanical energy without any loss. No lost energy flow such as thermal or acoustic energy is shown in the model. This reasoning is qualitative, since no quantitative information is needed to perform it and the inference also does not produce any quantitative insight about the design, such as exactly how much energy is to be lost. Using this type of reasoning, the modeler-inflicted inconsistency such as in Figure 3.4(a) could be reported back to the modeler.



### 3.5.3 Qualitative Modeling and Reasoning in Contemporary Representations

The Function Structure representation, commonly used to construct models within the Design Repository [50, 53] and the graph grammar-based synthesis tool [6], is not formalized enough at present to rigorously conclude if it supports quantitative modeling or not. Some models in the repository contain quantitative details of flows, although those details are merged within the flow names, since there is no other placeholder formally designated within the representation to capture quantitative detail. Thus it should be said that while the current level of formalism does not provide for quantitative information, the models can be and are used to capture such details by extending the formal scope of the representation.

Other function representations that are more formalized typically support qualitative modeling, although some provide placeholders for defining attributes or parameters of functions or flows. For example, the Functional Representation (FR) [61, 62] does not have any static vocabulary for the flow types or their attributes, but its implementation as the Causal Function Representation Language (CFRL) allows defining a device such as battery with parameters such as stored charge and electromotive force. However, these parameters are not designed to accept numeric values. Similarly, the Function-Behavior-State model [16, 17], the Structure-Behavior-Function model [12, 68], or the function ontologies [27-29, 64-66, 148] do not provide for quantitative modeling. Consequently, the reasoning performed on the respective models is primarily qualitative.

### 3.5.4 The Qualitative Modeling Requirement

**The representation must allow the designer to describe a design even when quantitative information is not available.** This is a fundamental requirement in the early, formative stages of design concepts designers may not have quantitative information to include in the model and an important objective of this representation is to allow computer-based modeling in the absence of such details. A design tool that requires quantitative information to add an object to the model, such as the geometric CAD tools that requires the necessary geometric information to draw a line (e.g., the terminal point coordinates or a point coordinate, a vector, and a length) would not be useful for this early design requirement.

An additional requirement on the modeling environment is that it should not be unnecessarily restrict the use to a linear modeling pattern. For example, it should allow the designer to develop invalid models and continue to develop a model even when the reasoning system identified a potential invalid modeling move, as the modeling sequence is a reflection of the designer's thought sequence, where not all advances in decision are made after attesting their validity. Often an advance is made to test its outcome beyond the immediate consequences. Thus, in order to facilitate exploration, the tool should allow intentional invalid modeling and progress without accepting reasoning suggestions. However, as mentioned earlier, this requirement applies more to the implementation of the representation in design tools, rather than the representation itself.

### **3.5.5 The Qualitative Reasoning Requirement**

**The representation must allow drawing qualitative inferences of two types from the models based on (1) conservation and (2) irreversibility.** The details of these reasoning types are discussed in Chapter 4. The representation addresses these requirements by incorporating the laws of conservation of mass and energy within the definitions in the function verbs at the first and the second layer of the representation. The subsequent layers composed of these concepts are then guaranteed to support both types of reasoning, as illustrated in Chapter 9.

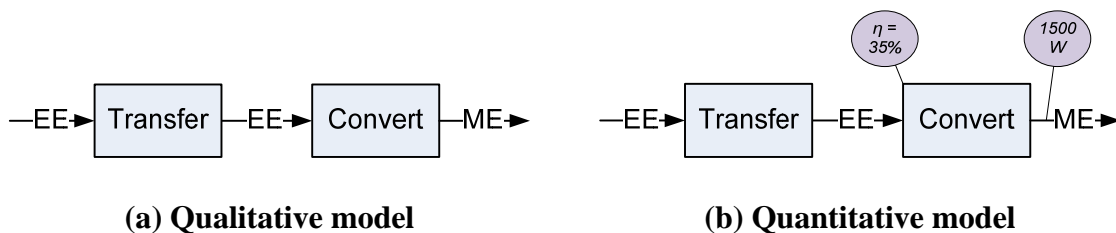
## **3.6 Extendibility**

### **3.6.1 What is Extendibility?**

Extendibility is described as a fundamental character of formal representations [31, 44] and describes how well the representation can accommodate addition of new functionality to meet newer requirements of representation and reasoning while minimizing the effort of redesign and implementation. Distinct from scalability, which relates to the ability to operate on a wide range of parameters such as complexity that were considered during designing the representation but were not tested over the entire range, extendibility is concerned with the ability to grow to meet newer requirements that were not considered during designing the representation. Similar to scalability, the extendibility requirements for a representation must be identified for each representation separately. For the representation in this research, one modeling extension and two reasoning extensions are considered: (1) quantitative reasoning, and (2) causal reasoning.

### 3.6.2 Quantitative Reasoning Extension

In the future, the representation should be able to describe quantitative details of a model and support reasoning using that additional quantitative information. Referring to Figure 3.5, which is redrawn here in Figure 3.6, the future extension should support assigning magnitudes to the physical parameters such as power of the mechanical energy flow or the efficiency of the functions. Moreover, it should support quantitative reasoning, for example, to predict the required electrical energy input based on that quantitative information.



**Figure 3.6: Qualitative and quantitative models**

In order to support this extension, the representation must allow adding placeholders for the quantitative attributes and magnitudes of the model elements (functions, flows). It is anticipated that this extension will not be directly applicable to the same early stage of design as addressed by this dissertation research. However, it will potentially be useful to build software tools that can span across design stages. For example, the same early design software could be used in the conceptual stage—where quantitative details are not extracted from a model—and the embodiment stage—where

the same models can be extended to include quantitative information, thus allowing a smooth transition and retention of design intent across design stages.

### 3.6.3 Causal Reasoning Extension

**In the future, the representation should be able to describe causal relations between functions and flows, in order to support physics-based causal description and predictive analysis of early design.** Causal reasoning is widely used in design for explanation generation [7, 61], failure prediction and propagation analysis [94-97, 144, 149]. However, it has not been performed using formal physics-based reasoning on graph-based function structures. As seen in Chapter 10, in some cases, causal deductions can directly use the conservation relations built within this representation, thus making this extension of particular interest. For more complex models, a separate description of causal relations between model elements that work in synch with the function representation may be needed. With this causal reasoning support, design tools built on this representation could potentially support failure analysis. For example, in Figure 3.6, it could be used to predict that if the electrical energy flow fails, it causes a failure in the output mechanical energy flow. When such causal derivations are topologically propagated through a function structure graph, much useful insight could be earned about the critical functional modules (weakest link), robustness (functional redundancy), or the spread of damage during a failure (propagation tree depth and width).

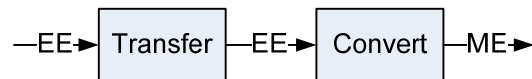
## **3.7 Scalability**

### **3.7.1 What is Scalability?**

Scalability is described as a fundamental character of formal representations [31] and describes how well the representation can solve problems within a given class that vary in size, complexity, and abstractness, thus indicating that it can handle “growing amounts of work” [150]. Representations are often designed and tested based on needs identified in sample problems in a domain and it is difficult to logically claim that the needs thus identified are the complete set that needs to be considered in design. For example, the vocabulary of function verbs in this dissertation research is developed and tested using sample problems in physics-based function modeling, shown in Chapter 9. While a large set of sample problems gives a reasonably high confidence about the representation’s ability to model any device, it is difficult, if not impossible to logically prove or disprove that claim. In lieu of an exhaustive set of samples problems used in design and a logical proof of coverage over all problems in the domain, the purpose of demonstrating scalability is to illustrate that the representation can model devices beyond the ones used to design it and that vary in different characters such as size and complexity. Scalability requirements are difficult to define for all representations at once and must be identified for each representation separately. The following discussion identifies the scalability requirement.

### 3.7.2 The Scalability Requirement

The representation should support modeling and reasoning on function structure graphs that vary in number of nodes and edges. Number of nodes and edges are well-accepted metrics of graph complexity in mathematics [49] and design research [151-156]. Theoretically, there is no limit on this graph complexity for the representation. Practically, this scaling can be limited by the computing resource requirement for storing the model elements, the algorithmic complexity of the reasoning algorithms, and the limitation of display capabilities for showing large models. For example, given a function structure, a designer may want to search for **all occurrences** of subgraphs consisting of Figure 3.7. This type of search may be used in searching solution principles that satisfy a specific subfunction module, such as converting EE to ME in this case, using solution principle catalogs that store solutions against the functions they perform [157, 158].



**Figure 3.7: Subgraph being searched within a function model**

Since **all instances** of the subgraph are to be found, a sequential search algorithm is preferred over faster algorithms such as the binary search, especially because the difference between their speeds is not significant for small search spaces (small function models). The algorithmic complexity of sequential search is proportional to the size of the list searched, given by  $O(N)$  in the big-O notation. In the case of searching for a subgraph, the algorithm has to proceed by first searching for all instances of EE, forming

a list of those instance pointers, then searching within that list for all instances where the EE flow is an input to a Transfer function, then looking for all instances of “EE-Transfer” that produce an outgoing flow of EE, and so on. Although there are some constraints imposed on function verbs definitions that can be used to reduce the number of comparisons in most cases, such as the equality of input and output flow types across Transfer, the worst case complexity is unaffected by those simplifications, as not all functions or subgraphs will benefit from them. Therefore, if there are  $F$  functions and  $f$  flows in the model, and there are  $i$  functions and  $j$  flows in the searched subgraph ( $i = 2, j = 3$  in this case), the big-O complexity of the search algorithm will be  $O(F^i \times f^j)$ . Thus, computation time for these subgraph pattern matching algorithms will increase exponentially with the size of the searched subgraph and may become a computation bottleneck when large subgraphs are searched within large models.

However, typical reasoning algorithms for checking model consistency with conservation laws will have lower complexity. For example, the complexity for checking if the functions in a model obeys conservation laws is  $O(F \times i \times j)$ , where  $F$  is the number of functions,  $i$  is the average number of input flows to a function, and  $j$  is the average number of output flows. Thus, the degree of connectedness of the graph ( $i, j$ ) is also important in algorithmic complexity.



The range of model size within a sample of eleven function structure graphs used in previous research [159] out of the 110 in the Design Repository<sup>10</sup> is between 17 and 50 for function count and between 30 and 104 for flow count. **Based on this data, the scaling requirement is that the representation must support modeling and reasoning on function structures with up to 50 functions and 150 flows, without causing perceptible memory overflow or slowness.**

### 3.8 Consistency and Validity

Consistency or internal consistency is the quality of formal logic systems that ensures lack of contradiction within the statements of the system [109, 111, 113, 114, 160]. **It should be impossible to infer two statements P and Q through logical deductions from the declarations made within the representation—such as the definitions of verbs, nouns, relations, constraints, and attribute definitions—such that  $P = \neg Q$ .** Consistency is an internal property of a representation, as it does not require that the statements producible by the representation be true according to external knowledge. It only requires agreement between statements derived within the representation. Consistency of the proposed representation is demonstrated in Section 6.1.

---

<sup>10</sup> Based on June 2009 data

Validity or external validity of a logic system against an external body of rules is the quality to ensure that starting from a premise that is true according to the external rules, it is impossible to logically derive an inference that is false according to the same external rules using declarations within the logic system and their implications [160]. **The representation must be valid against the principles of conservation and irreversibility. Specifically, if a model implies a violation of any of these principles, the representation should support a reasoning algorithm that can detect that violation.** For the representation in this dissertation, it means that when the representation used to model a real mechanical device, none of the implications of the model makes a violation of the laws of conservation or irreversibility. In contrast with consistency, validity is an external property of a representation, as it checks for agreement between the implications of statements within the representation and an external body of rules. For this representation, the external rules are the laws of conservation of mass and energy, and the principle of irreversibility. Validity of the representation against the conservation principle is demonstrated in Section 6.2, while its validity against the irreversibility principle is demonstrated in Chapter 7.

In summary, this chapter elaborates on the requirements the formal representation must satisfy in order to support function-based formal analytical reasoning in early design. In the next chapter, specific reasoning needs for this representation are identified through a simulated modeling exercise. From this exercise, the algorithmic steps for reasoning and the information elements necessary to capture in the representation are identified.

## CHAPTER 4. SYSTEMATIC DISCOVERY OF REASONING NEEDS AND INFORMATION ELEMENTS FOR THE REPRESENTATION

The information elements necessary to support reasoning on conservation and irreversibility are identified in this chapter in three steps. First, specific reasoning tasks under these categories are identified through a modeling exercise (Section 4.1). Next, procedures are developed to perform those tasks algorithmically (Section 4.2). Finally, these procedures are inspected for extracting information elements (Section 4.3). These information elements and the reasoning tasks they support define the data and reasoning requirements for the representation, developed in the next chapter.

### **4.1 Discovering Reasoning Needs: The Chalkboard Exercise**

The purpose of this exercise is to discover the type of analytical reasoning a designer may use or want to receive through a designer-model conversation during new product design. This conversation is analogous to a sketch-designer conversation discussed in previous research [15]. In the exercise, a designer develops and explores the functional architecture of an electromechanical design product as part of concept development, while verbal feedback is provided to simulate computer-aided analysis feedback on the in-process model states.

#### **4.1.1 Design Problem Selection**

The design artifact is one that the designer had used earlier and of which he understands the working principles, but one that he had not examined or designed

previously. Such a product ensures that the designer is able to ideate and develop a reasonably detailed function model, but is unlikely to be biased by previous familiarity with the component-level details of the product. The following design problem is given.

### The Design Problem

*“Design an air-heating device that intakes air from one location in a house and delivers hot air to another location. The device should consume approximately 3 kilowatts of power”.*

Three high-level requirements from Chapter 3 are used in selecting the problem and the simulated reasoning feedback.

1. **Domain coverage** (Section 3.1): The design problem is chosen to involve electrical, mechanical, thermal, and fluid phenomena, and possible use of acoustic, optical, or other principles, thus necessitating the designer to use principles in a variety of domains.
2. **Normative modeling** (Section 3.4): The designer is given a problem that is new to him, with instructions to describe the intended functionality of the device. The designer never designed this or similar systems previously and has no work experience in the HVAC domain. The descriptive modeling requirement is not directly addressed in this exercise. It is later used to test domain coverage of the vocabulary of the new function vocabulary in Chapter 7.

3. **Qualitative modeling and reasoning** (Section 3.5): The designer is instructed to keep the model qualitative initially (models in Section 4.1.4) and add quantitative details only when he thinks that the qualitative model describes the functions correctly (Section 4.1.5). In order to simulate non-restrictive modeling, the designer is instructed to optionally respond to or ignore the suggestions produced by the tool. All feedbacks are provided in qualitative form, for as long as the model itself is qualitative, thus simulating qualitative reasoning.

Some of the high-level requirements from Chapter 3 are not considered in this exercise. The domain-independence and physics-based concreteness requirements are not addressed. Rather, these needs are discovered from this exercise as modeling needs necessary to support qualitative reasoning (Section 4.3). The descriptive modeling requirement is not addressed due to the inherently normative character of the design task. Rather, descriptive modeling is used in Section 9.1 during vocabulary testing. Scalability pertains to data structure design and cannot be addressed unless the information elements to be stored are first identified through this exercise.

#### **4.1.2 Participant Selection**

The participant in this exercise is a graduate student involved in function-based design research. He is academically trained in function modeling through a design theory and methods class. He designed and built electromechanical devices and tools in industry and academic projects. He used and designed geometric CAD and CAD-automation

software professionally for eleven years total. Because of this background, this participant is deemed suitable, as he could use both his product design and function modeling background during modeling and simulate model states that would illustrate the reasoning requirements effectively.

Only one participant is used in this exercise as the objective is not to discover a replicable trend of participant behavior or to discover reasoning needs. At the point of conducting this study, the need to support the three reasoning categories is already identified through literature review on functional reasoning. The reasoning messages passed to the designer are not produced by the participant or the model; they are provided from outside. The objectives of using the participant are (1) to articulate the exact tasks under the three pre-identified categories and (2) to verify that a designer could use those reasoning helps in developing a model, when supplied by a computer.

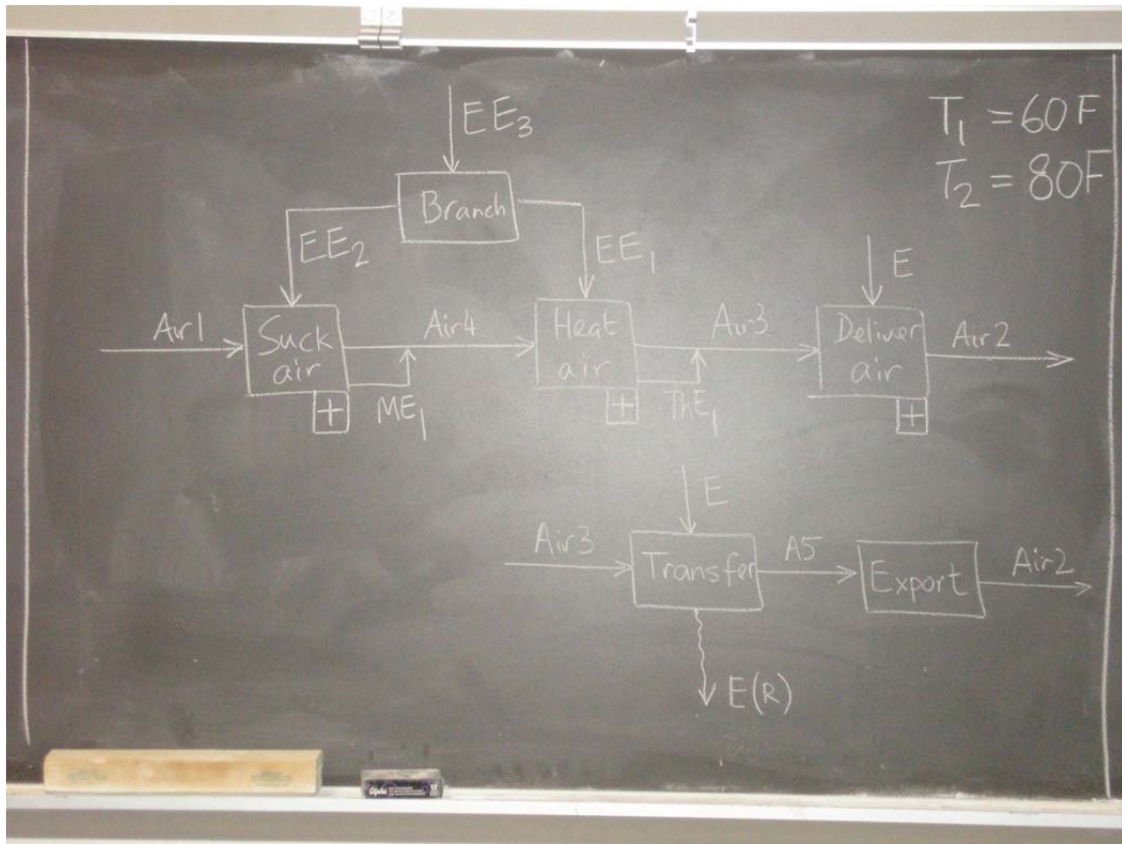
#### **4.1.3 Modeling Interface and Feedback**

The modeling interface is a chalkboard inside a design lab, which gives the designer a familiar work environment and the flexibility for easy erasing and editing, thus creating an environment suitable for creative tasks. During this modeling session, a “conversation” between the designer and the model is simulated by supplying verbal feedback messages to the designer, between modeling steps. These messages represent results of analytical reasoning performed by an ideal function-modeling software tool that hypothetically replaces the chalkboard. Care is taken to ensure that the messages only provide analytical feedback, rather than synthesis directions, and that they can be

produced purely based on two types of knowledge: (1) **conservation** laws and (2) the **irreversibility** principle. The synthesis of the model is controlled by the designer. The tool does not suggest what the designer should do. It only keeps the growth of the model controlled by checking it against these laws that inevitably apply to any concept.

#### **4.1.4 Exercise Steps: Black Box Modeling (Qualitative Reasoning)**

The modeling steps are captured in photographs (Appendix A), which reconstruct the modeling history similar to a storyboard, when viewed serially. One step from this sequence is shown in Figure 4.1 for reference. This figure is the 27<sup>th</sup> step of a total 44-step process and shows an intermediate state of the model where the designer identifies several functions and flows.



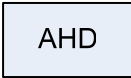
**Figure 4.1: A sample step from the chalkboard modeling exercise**

To illustrate reasoning, only the salient steps of the process are reproduced in Table 4.1 through Table 4.8. The reasoning depends only on the state of the model and not on the modeling process. Thus omitting these interim steps does not impact the reasoning discovery. The shown steps are in their original order of construction. For each state, the feedback supplied is mentioned with its rationale. This section shows steps leading up to the black box model and demonstrates the use of **conservation** and **irreversibility** reasoning at a **qualitative** level. Two types of conservation reasoning are identified in the Message section of the tables: (1) topologic and (2) derivational. The first uses only topological connectedness between model elements to generate the

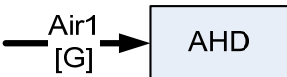


messages, while the second actually uses the conservation laws. Extension of these reasoning types at the quantitative level is illustrated in the next section.

**Table 4.1: Redundant function inference**

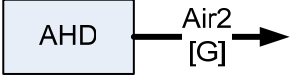
 <p><b>Model State 4.1: Function instance AHD (Air Heating Device)</b></p>	
<b>Message</b>	Topologic inference: Redundant function: AHD
<b>Rationale</b>	A function that has no input or output flow attached is considered a redundant function, as it is not performing any transformative action and is therefore not contributing to the overall functionality of the model. While the function is perhaps drawn with the intent of adding flows to it next, at the present state, it matches the description of a redundant function.

**Table 4.2: Dangling tail and barren flow inference**

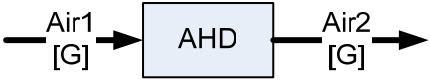
 <p><b>Model State 4.2: AHD with one input flow instance Air1 (type: Gas)</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Topologic inference: Dangling tail: Air1</li> <li>2. Derivational inference: Barren flow: Air1</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. Every flow must have at least one “parent flow” from which it is derived, and at least one “derivative flow” that is derived from it, as</li> </ol>

	<p>otherwise, either its genesis or its outcome is unexplained, which violates <b>conservation</b>. The only exception to this rationale is in the case of the flows that are derived from singularity nodes, such as the environment and functions for storing and supplying flows, as explained in Section 8.2. Air1 does not have any entity—function or flow—at its tail to explain its genesis.</p> <p>2. Although the head of Air1 is not dangling (attached to AHD), the model does not show a <b>derivative</b> of Air1 and AHD is not marked as a singularity. This combination is another violation of <b>conservation</b>.</p>
--	---

**Table 4.3: Dangling head and orphan flow inference**

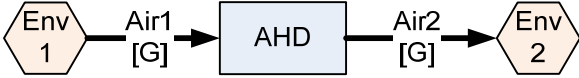
	
<p><b>Model State 4.3: AHD with one output flow instance Air2 (type: Gas)</b></p> <p>This model is produced by an alternate modeling sequence from Model State 4.1.</p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Topologic inference: Dangling head: Air2</li> <li>2. Derivational inference: Orphan flow: Air2</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. Air2 does not have any entity—function or flow—to explain its derivation.</li> <li>2. Although the tail of Air2 is not dangling (attached to AHD), the model does not show a parent of Air2 and AHD is not marked as a singularity.</li> </ol>

**Table 4.4: Material transformation without energy exchange**

 <p style="text-align: center;"><b>Model State 4.4: AHD with two material flows</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Topologic inference: Dangling tail: Air1; Dangling head: Air2</li> <li>2. Derivational inference: Air1 → Air2 (“→” reads as “is derived as”)</li> <li>3. Derivational inference: Transformation of material Air1 to Air2 without energy transaction</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. The orphan and barren flows are abolished, since a derivational relation can be inferred. However, the dangling tail and head messages are still valid.</li> <li>2. If there is only one input and only one output flow attached to a function and they are of the same major type (Material, Energy), the output must be derived from the input to satisfy conservation (<b>one-in-one-out</b>). If they are of different major types, such as one a Material and the other an Energy, this inference cannot be drawn, since conversion between material and energy is not accepted.</li> <li>3. Any change in material state requires exchange of energy to or from the material, as a consequence of the first law of thermodynamics [128, 129]. This exchange does not always imply “consumption”, as in some cases energy may be released, such as the cooling of a hot</li> </ol>

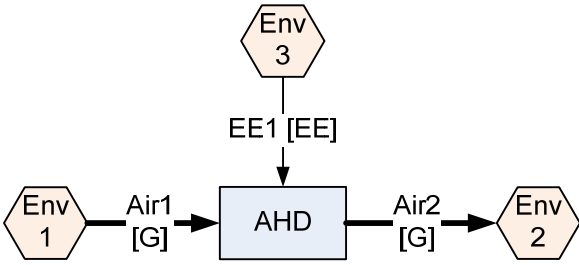
	<p>metal plate in air, which liberates heat. Alternately, the increase of temperature of air could be realized without exchange of energy with the surroundings, such as by converting one form of its energy into raising its internal energy (manifested as temperature). For example, the kinetic energy of Air1 could be reduced to cause the difference to manifest as internal energy. However, using the closed-world assumption, it is assumed that unless such a relation is explicitly mentioned, it is not true. No exchange of energy is shown to or from Air1 or Air2. Thus, the model must be faulty to show that temperature changes between the two flows.</p>
--	--

**Table 4.5: Environments as singularity nodes**

	
<p><b>Model State 4.5: Flow instances connected to environment instances</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air1 → Air2</li> <li>2. Derivational inference: Transformation of material Air1 to Air2 without energy transaction</li> </ol> <p><b>Note:</b> The dangling tail and head messages are withdrawn.</p>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. The inference is explained in the previous steps.</li> <li>2. The first law violation reasoning still applies from the previous step.</li> </ol> <p><b>Note:</b> Air1 is introduced from the Environment Env1 and Air2 is dismissed</p>

to Env2. For model-level conservation consistency, Environment instances are acceptable as singularities, similar to sources and sinks in thermodynamic processes [128, 129]. Hypothetically, they can supply or receive indefinite amounts of material or energy, without undergoing any state change (transformative action). Thus, Environments are not functions, although they are nodes in the function structure graph. They define the functional scope of the model. For example, as long as a flow is submitted to the environment, the model does not need to show or support reasoning on its further derivations. In reality, the flow may undergo further transformations, but those are outside the model's scope.

**Table 4.6: Unused (barren) energy flow**

	
<p><b>Model State 4.6: Energy flow EE1 input to support material transformation</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air1 → Air2</li> <li>2. Derivational inference: Barren flow: EE1</li> <li>3. Derivational inference: Transformation of material Air1 to Air2 without energy transaction</li> </ol>

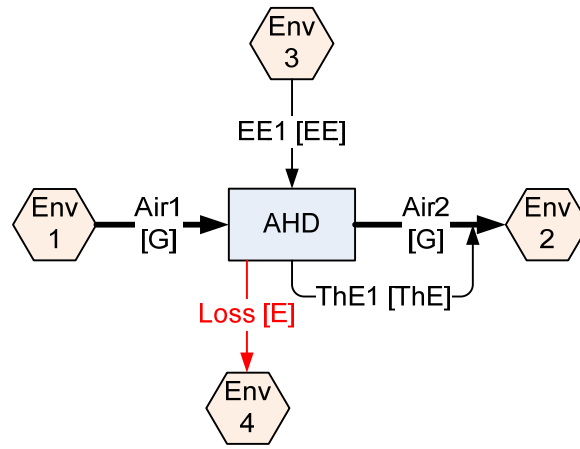
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. The derivation inference is explained in previous models.</li> <li>2. EE1 is a barren flow, as its derivative is not shown in the model.</li> <li>3. The third message continues, since although EE1 is input to the function, there is no indication of energy exchange to or from the air flows.</li> </ol>
------------------	---

**Table 4.7: Carrier flow and irreversibility inference**

<p style="text-align: center;">Air2.T &gt; Air1.T</p>	
<b>Model State 4.7: EE1 transformed into ThE1 and added to Air2</b>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air1 → Air2</li> <li>2. Derivational inference: EE1 → ThE1</li> <li>3. Irreversibility inference: Conversion of energy without any loss: 100% efficiency implied in AHD.</li> </ol> <p><b>Note:</b> The message about EE1 being a barren flow is withdrawn.</p> <p><b>Note:</b> The message about material transformation without energy transaction is now withdrawn.</p>

<b>Rationale</b>	<ol style="list-style-type: none"><li>1. The derivation inference is explained in previous models.</li><li>2. The second derivational message is explained similar to the first. If a function has only one energy input and only one energy output, there is no other way but form them to be in a parent-derivative relation, in order for conservation to satisfy. Additionally, the two energy flows are also of different types, which ensure that the function AHD is not classified as ineffective.</li><li>3. Any energy transforming process must operate at efficiency less than unity and incur losses, as a consequence of the second law of thermodynamics. However, the notion of loss relies on the notion of what is useful and what is unwanted, as illustrated in Table 9.1 later. For example, heat produced in a light bulb is considered loss since light is the sought form of energy. This notion inverts in the case of a heat lamp, where heat is sought and light is a loss. Thus, it is required that every transformation of energy produces at least two output energy flows, one of which may be a loss. The two flows may be of the same type, in which case loss represents the portion of output energy that cannot be used in a subsequent function. For example, the model in this state implies the incoming electrical energy is transformed into heat in entirety, while in reality a portion of the heat (same type as ThE1) would be lost because it escapes without being added to Air2.</li></ol>
------------------	---

**Table 4.8: One-in-many-out derivation (acceptable black box model)**



**Model State 4.8: Lost energy included in model: An acceptable model**

<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air1 <math>\rightarrow</math> Air2</li> <li>2. Derivational inference: EE1 <math>\rightarrow</math> {ThE1, Loss}</li> <li>3. Acceptable model state.</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. The first derivation message is explained in previous steps.</li> <li>2. When there is only one input flow of a major type (Material, Energy) and more than one output flow within that major type, conservation requires that all the output flows be derived from the single input flow of that type (<b>one-in-many-out</b>).</li> <li>3. The 100% efficiency message is addressed by modeling a flow of lost energy. The type of this flow is intentionally kept generic to Energy (E), as the designer is unsure about the specific forms of loss at this early stage. Overall, this model agrees with the laws of conservation.</li> </ol>



In total, the above modeling steps identify eight distinct reasoning tasks:

1. Redundant function (Model State 4.1)
2. Dangling tail (Model State 4.2)
3. Dangling head (Model State 4.3)
4. Barren flow (Model State 4.2)
5. Orphan flow (Model State 4.3)
6. One-in-many-out derivation inference (Model State 4.8)
7. Material transformation without energy (Model State 4.4)
8. Missing residual flow (Model State 4.7, Model State 4.8)

Each task is based on either conservation or irreversibility principles. This modeling session is continued next to reveal reasoning at a quantitative level.

#### **4.1.5 Exercise Steps: Model Decomposition (Quantitative Reasoning)**

This section describes a continuation of the above modeling exercise to illustrate **quantitative** reasoning. In this exercise, the designer decomposes the model by referring to the black box, using a separate work area on the chalkboard. This choice of a separate work area parallels to saving the black box model to the disk and using a new workspace to perform decomposition. These modeling steps are illustrated in Table 4.9 through Table 4.14. In each table, the message and the rationale are illustrated for a model state.

**Table 4.9: Topologic inferences during decomposition**

<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 5px; margin: 5px;">Draw Air</div> <div style="border: 1px solid black; padding: 5px; margin: 5px;">Heat Air</div> <div style="border: 1px solid black; padding: 5px; margin: 5px;">Deliver Air</div> </div> <p style="text-align: center;"><b>Model State 4.9: User-driven decomposition (first level)</b></p>	
<b>Message</b>	1. Topologic inference: Redundant function: Draw Air, Heat air, Deliver Air
<b>Rationale</b>	1. The redundant function message is explained in Model State 4.1

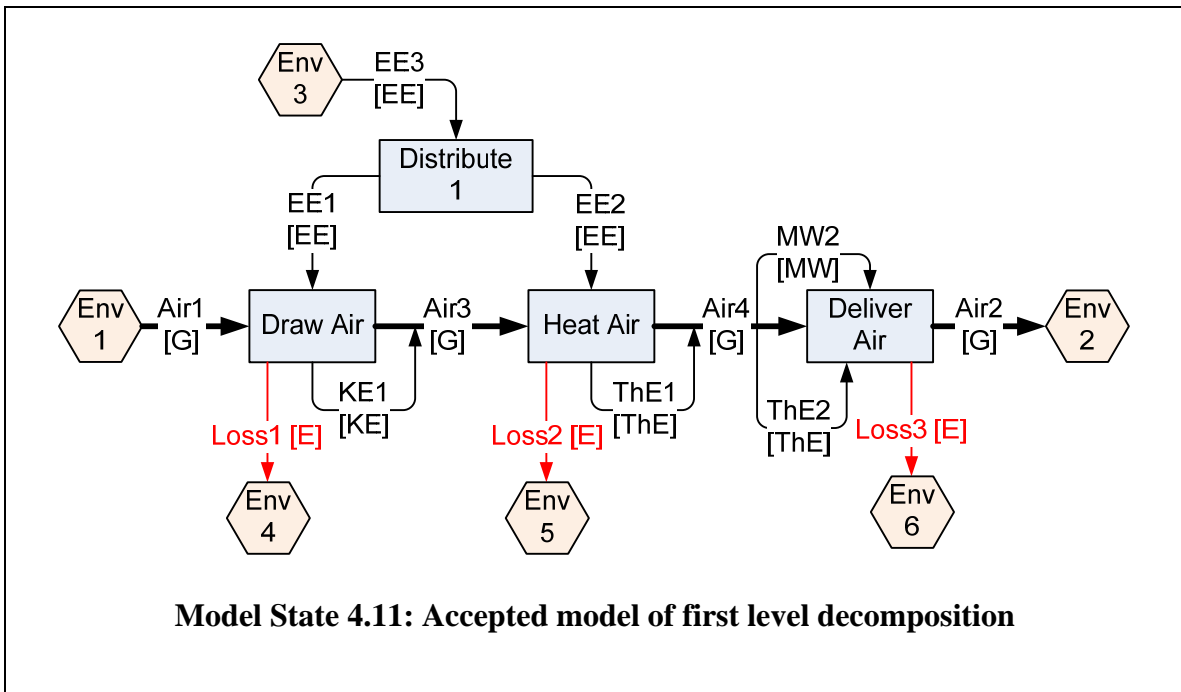
**Table 4.10: Material transformation without energy in a decomposed model**

<p style="text-align: center;"><b>Model State 4.10: Stream of air flow through functions</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air1 → Air3 → Air4 → Air2</li> <li>2. Derivational inference: Transformation of material without energy exchange: Draw Air, Heat Air, Deliver Air</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. In derivation inference is explained in Model State 4.4. The observation of interest is that the elementary reasoning actions identified in the black box modeling session in Section 4.1.3, such as derivation inference, are now used to compose more complex inferences.</li> <li>2. The energy exchange message is discussed in Model State 4.5</li> </ol>

By definition of functions and flows (Section 1.2), two flow instances have to be different at least by their state and a function must cause a change of state or type of a flow, otherwise it is an ineffective function. In this modeling exercise, the simulated software assumes that different instances of the same flow type in a model must be of different states. For example, Air1 is the air outside the inlet orifice of the device, which is at a lower velocity (typically zero) than Air3, which represents air after being operated by the Draw Air function and flowing faster than Air1. From Air3 to Air4, the major change is in temperature, indicated in the second relation.

The following model state shows a further decomposed version of the model, without walking through the steps leading to it from Model State 4.10, since those steps do not illustrate any new reasoning. Model State 4.11 is the end product of the first level of decomposition and illustrates how the designer planned energy exchange through the product to satisfy conservation, leading into the second level of decomposition.

**Table 4.11: Many-in-one-out inference (Accepted model)**



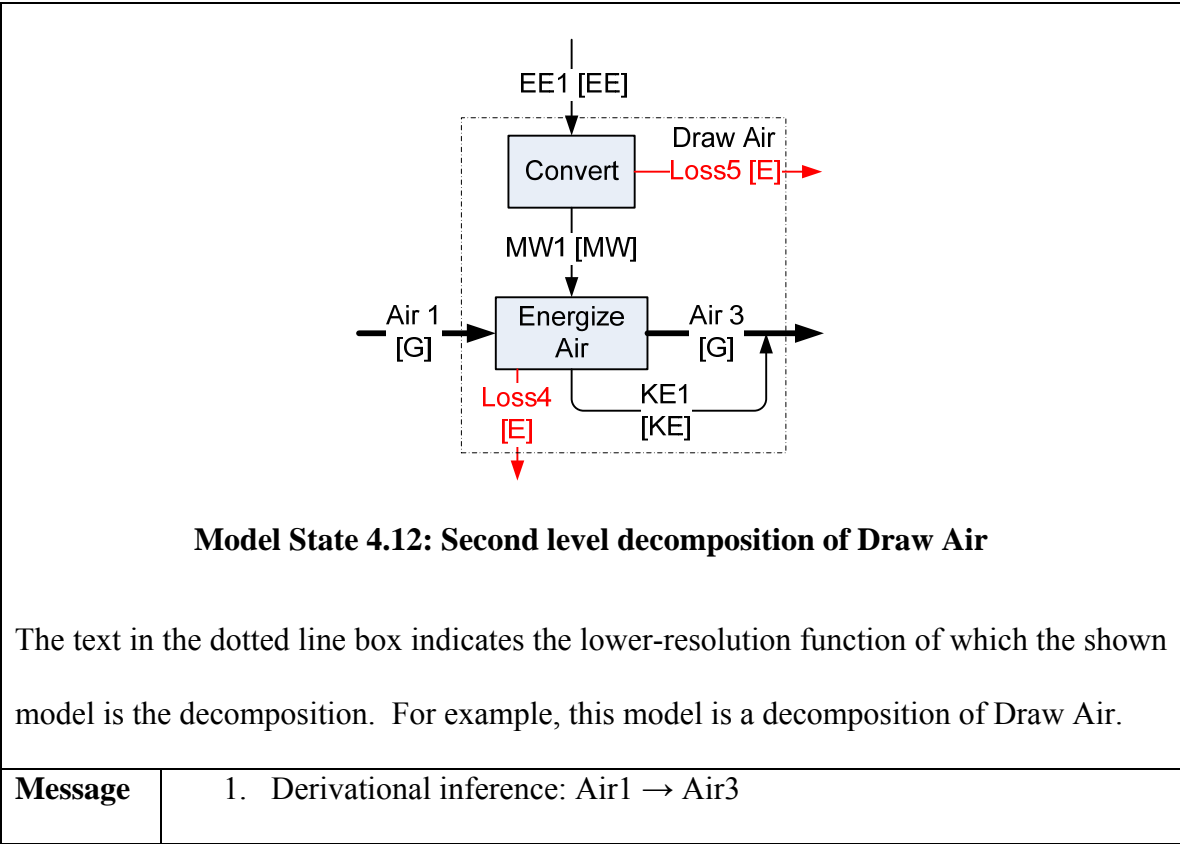
**Model State 4.11: Accepted model of first level decomposition**

<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: <math>Air1 \rightarrow Air3 \rightarrow Air4 \rightarrow Air2</math></li> <li>2. Derivational inference: <math>EE1 \rightarrow \{KE1, Loss1\}</math>, where KE is kinetic energy</li> <li>3. Derivational inference: <math>EE2 \rightarrow \{ThE1, Loss1\}</math></li> <li>4. Derivational inference: <math>EE3 \rightarrow \{EE1, EE2\}</math></li> <li>5. Derivational inference: <math>\{MW2, ThE2\} \rightarrow Loss3</math>, MW is mechanical work</li> <li>6. Acceptable model state</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1-4. The derivation inferences are explained in the previous steps.</li> <li>5. When there are more than one input flow of a major type (Material, Energy) and only one output flow within that major type, all the input</li> </ol>

	flows must be conserved as the single output flow ( <b>many-in-one-out</b> ).
	6. The model only returns inferred derivations and does not imply an error.

The above steps show the exploration of the first level of decomposition, which ends with an acceptable state. Beyond this point, the designer explores deeper into the subfunctions identified above, in order to resolve them to more well-defined terms that can support solution search and physics-based reasoning.

**Table 4.12: Flow preservation and additive inference in decomposition (Draw Air)**



	<ol style="list-style-type: none"> <li>2. Derivational inference: <math>EE1 \rightarrow \{MW1, Loss5\}</math></li> <li>3. Derivational inference: <math>MW1 \rightarrow \{KE1, Loss4\}</math></li> <li>4. Resolution inference: Accepted preservation of boundary flows</li> <li>5. Resolution inference: <math>Loss4.E + Loss5.E = Loss1.E</math></li> <li>6. Accepted model state</li> </ol>
<p><b>Rationale</b></p>	<ol style="list-style-type: none"> <li>1-3. The derivational inferences are explained in previous model states.</li> <li>4. The decomposition action is valid, since the flows attached to the Draw Air function before decomposition is accounted for at the overall boundary of the decomposed model. This quality of decomposition is named here the <b>preservation of boundary flows under decomposition</b>.</li> <li>5. The loss flows Loss4 and Loss5 in the decomposed model replace the single loss flow Loss1 in the composed model. Thus, the energy content of Loss4 and Loss5 must equal that of Loss1. This inference does not indicate a derivational relation that Loss1 is derived into Loss4 and Loss5. Rather, it implies that the modeled Loss1 flow at a lower resolution is the sum of the two loss flows at the higher resolution model. This type of inference is called here the <b>additive inference across decomposition levels</b>.</li> <li>6. The model it obeys the balance laws of mass and energy, and accounts for irreversibility.</li> </ol>

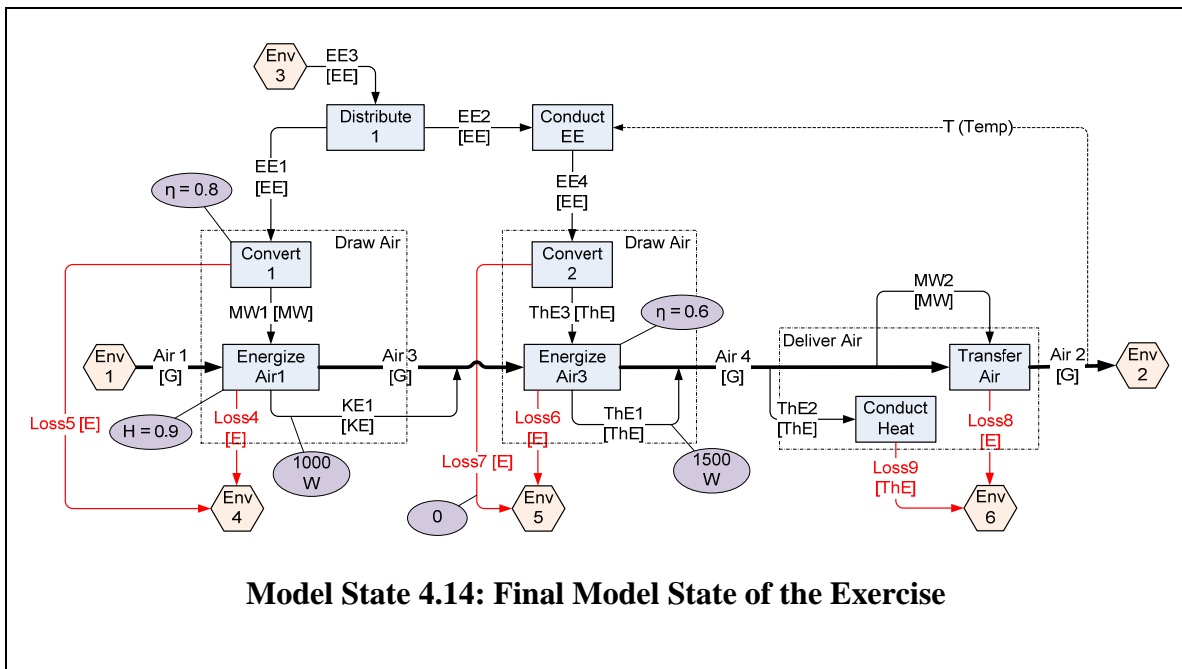
**Table 4.13: Flow preservation inference in decomposition (Deliver Air)**

<p><b>Model State 4.13: Second level decomposition of Deliver Air</b></p>	
<b>Message</b>	<ol style="list-style-type: none"> <li>1. Derivational inference: Air4 → Air2</li> <li>2. Derivational inference: MW2 → Loss6</li> <li>3. Derivational inference: ThE2 → Loss7</li> <li>4. Resolution inference: Accepted preservation of boundary flows</li> <li>5. Resolution inference: Loss8.E + Loss9.E = Loss3.E</li> <li>6. Accepted model state</li> </ol>
<b>Rationale</b>	<p>All of the reasoning messages are similar to those explained in previous steps.</p>

The final model state of the modeling exercise is shown in Model State 4.14 below, where the designer adds quantitative details. He decides that the conversion of EE4 to ThE3 happens without loss, since the device is probably a resistive heater that converts all of the consumed EE into heat. Energize\_Air3 has an efficiency of 0.6, since 40% of the produced heat is lost through the insulation. The total heat added to air is

1500 Watts. The designer may know this value from external calculations or may enter it only to explore a what-if scenario using the model. He estimates that 1000 Watts are necessary to cause the air to flow, potentially using a blower. The blower blades have an efficiency of 90%, while the blower motor rejects 20% of its input energy as heat (Loss5). Based on this data, the designer wants to explore the total power required to operate the machine.

**Table 4.14: Quantitative reasoning on efficiency and power required**



**Model State 4.14: Final Model State of the Exercise**

<b>Message</b>	<ol style="list-style-type: none"> <li>1. All inferences from Model State 4.11, Model State 4.12, and Model State 4.13</li> <li>2. Derivational inference: <math>EE2 \rightarrow EE3 \rightarrow \{ThE3, Loss5\}</math></li> <li>3. Total power required = 3889 Watts</li> </ol>
<b>Rationale</b>	<ol style="list-style-type: none"> <li>1. The inferences carried forward from the previous model states are</li> </ol>



	<p>explained in the respective model state rationales.</p> <p>2. The conduct function uses only one input and one output energy, and thus the inference must hold. Moreover, the conduct function receives a signal from the temperature attribute of the existing air flow Air2.</p> <p>3. The power required is computed as follows.</p> $\text{ThE3.E} = \frac{\text{ThE1.E}}{\eta_{\text{En\_Air3}}} = \frac{1500\text{W}}{0.6} = 2500\text{W}$ $\therefore \text{EE2.E} = \text{EE4.E} = (\text{ThE3.E} + \text{Loss7.E}) = (2500 + 0)\text{W}$ $\text{EE1.E} = \frac{\text{MW1.E}}{\eta_{\text{Convert1}}} = \frac{(\text{KE1.E} / \eta_{\text{En\_Air1}})}{\eta_{\text{Convert1}}} = \frac{(1000\text{W} / 0.9)}{0.8} = 1389\text{W}$ $\therefore \text{EE3.E} = (\text{EE1.E} + \text{EE2.E}) = (2500\text{W} + 1389\text{W}) = 3889\text{W}$
--	---

By comparing this final model (Model State 4.14) with the black box (Model State 4.8) and the first level decomposition (Model State 4.11), the evolution and boundary flow preservation under decomposition can be verified. It also illustrates how model decomposition helps in understanding design problems. In this case, although the problem statement mentions delivery of air, the designer is focused on the main function of heating at the black box level and shows only the addition of thermal energy (ThE1) to the air. It is only when the inner details of the device are planned that the need for driving the air is identified, which resulted into adding kinetic energy KE1 to the air flow in the decomposed model. Similar discovery of functionality happened with the energy exchange in the Deliver Air function, which was not completely predictable in the black box. The model decomposition exercise identifies four additional reasoning tasks:

1. Many-in-one-out derivation (Model State 4.11)
2. Boundary flow preservation in decomposition (Model State 4.12)
3. Additive inference across decomposition levels (Model State 4.12)
4. Quantitative inference about power required (Model State 4.14)

In all, twelve reasoning tasks are identified from the black box and decomposition exercises, with which a designer could be assisted while exploring function architecture of novel design. These reasoning tasks are summarized in Table 4.15. For each task, the table includes the model states where it is illustrated and the algorithm number where it is presented in the next section. The conservation reasoning tasks are divided into (1) topologic and (2) derivational, as explained in the next section. These algorithms are then inspected for information elements in Section 4.3.

**Table 4.15: Summary of Reasoning Needs Discovered**

<b>Reasoning</b>	<b>#</b>	<b>Reasoning Name</b>	<b>Model State</b>	<b>Algorithm</b>
Conservation: Topologic	1	Redundant function	Model State 4.1 Model State 4.9	Algorithm 4.1
	2	Dangling tail	Model State 4.2 Model State 4.4	Algorithm 4.2
	3	Dangling head	Model State 4.3 Model State 4.4	Algorithm 4.3
Conservation: Derivational	4	Barren flow	Model State 4.2 Model State 4.6	Algorithm 4.4

<b>Reasoning</b>	<b>#</b>	<b>Reasoning Name</b>	<b>Model State</b>	<b>Algorithm</b>
	5	Orphan flow	Model State 4.3	Algorithm 4.5
	6	One-in-one-out and one-in-many-out derivation	Model State 4.4 Model State 4.5 Model State 4.6 Model State 4.7 Model State 4.8 Model State 4.10 Model State 4.11	Algorithm 4.6
	7	Many-in-one-out derivation	Model State 4.11	Algorithm 4.7
	8	Material transformation without energy input	Model State 4.4 Model State 4.5 Model State 4.6 Model State 4.10	Algorithm 4.8
Irreversibility	9	Missing residual flow	Model State 4.7	Algorithm 4.9
Quantitative	10	Power required	Model State 4.14	Algorithm 4.10
Resolution Reasoning	11	Boundary flow preservation in decomposition	Model State 4.12	Algorithm 4.11
	12	Additive inference	Model State 4.12	Algorithm 4.12

Reasoning	#	Reasoning Name	Model State	Algorithm
		across decomposition levels		

## 4.2 Reasoning Algorithms

This section is presented with two goals: (1) to ascertain that the reasoning tasks identified in the modeling exercise can be indeed performed algorithmically and (2) to identify the information elements required to support such algorithms. The algorithms for the twelve reasoning tasks under the five types (Table 4.15) are presented in the tables below (Algorithm 4.1 through Algorithm 4.12). Commented lines are provided in red to help interpretation and therefore individual tables are not discussed.

### 4.2.1 Conservation Reasoning Algorithms (Topologic)

These three algorithms [115] classified as topologic reasoning, as they are written entirely based on the valid connections between functions and flows, without direct reference to the laws of conservation.

#### Algorithm 4.1: Algorithm for redundant function

```

Loop through FunctionInstanceList; // List of function instances in the
model
IF (FunctionInstanceList[i].InList.IsEmpty() &&
FunctionInstanceList[i].OutList.IsEmpty())
Echo "Redundant function: " + FunctionInstanceList[i].GivenName;

```

```
// InList and OutList are linked lists of flow pointers and are used to
store pointers to the input and output flow instances to a function
instance.

// IsEmpty is a Boolean member function in the linked list data
structure that returns TRUE if the list is empty.

//GivenName is a string data member that holds the name of an element.
```

#### **Algorithm 4.2: Algorithm for dangling tail**

```
Loop through FlowInstanceList;      // List of flow instances in the
model
IF (FlowInstanceList[i].pTailNode == NULL)
Echo "Dangling Tail: " + FlowInstanceList[i].GivenName;

// pTailElement is a pointer to the CElement instance at the tail of a
flow instance, which can be a function, an environment, or another
flow.
```

#### **Algorithm 4.3: Algorithm for dangling head**

```
Loop through FlowInstanceList;      // List of flow instances in the
model
IF (FlowInstanceList[i].pHeadNode == NULL)
Echo "Dangling Head: " + FlowInstanceList[i].GivenName;
```

```
// pHeadElement is a pointer to the CElement instance at the head of a  
flow instance, which can be a function, an environment, or another  
flow.
```

#### 4.2.2 Conservation Reasoning Algorithms (Derivational)

These algorithms are used to check a model's adherence to the laws of conservation of mass and energy. Since energy conservation is the essence of the first law of thermodynamics, these are also called the First Law Algorithms. The orphan and barren flow algorithms essentially enforce that a flow must be conserved or derived into other flows of the same major type (Material, Energy), as otherwise its outcome is unaccounted for. The one-in-many-out and many-in-one-out inference algorithms rely on separate conservation of mass and energy. Conservation of the mass-energy combination, such as in nuclear reactions where mass is changed into energy, is not allowed in this representation and will in fact return as a violation of the individual conservation laws. This simplification prevents reasoning on those reactions, as the outcome of the mass difference converted into energy and the genesis of the energy from mass are both unaccountable in this scheme. However, this intentional compromise allows for reasoning in other simpler designs. If the algorithm was written to check for the mass-energy combination, functions that input mass but outputs only energy (no mass) would not be detected as violation, as it would be explicable that the mass was converted to energy. However, this detection is important, as illustrated in the modeling exercise.

#### Algorithm 4.4: Algorithm for barren flow

```
Loop through FlowInstanceList;
IF ((FlowInstanceList[i].ChildList.IsEmpty) &&           // No child of
flow
(EnvList.Find(FlowInstanceList[i].pHeadNode) == NULL))
// the head of the flow is not going to an environment instance
Echo "Barren flow: " + FlowInstanceList[i].GivenName;

// ChildList is a list in the flow class used to store pointers to the
child flows of the flow object.  A flow can have more than one child.
// A flow that has no children (derivatives) and whose head is
connected to an environment instance is not to be detected as barren,
as the flow's head is outside the system boundary where conservation
rules do not apply.
```

#### Algorithm 4.5: Algorithm for orphan flow

```
Loop through FlowInstanceList;
IF ((FlowInstanceList[i].ParentList.IsEmpty) &&           // No parent of
flow
(EnvList.Find(FlowInstanceList[i].pTailNode) == NULL))
// the tail element of the flow is not an environment instance
Echo "Orphan flow: " + FlowInstanceList[i].GivenName;
```

```

// ParentList is a list in the flow class used to store pointers to the
parent flows of the flow object. A flow can have more than one parent.
// EnvList is a linked list containing pointers to all environment
instances in the model
// A flow that has no parents and whose tail is connected to an
environment instance is not to be detected as orphan, as the flow's
tail is outside the system boundary where conservation rules do not
apply.

```

**Algorithm 4.6: Algorithm for one-in-many-out derivation inference**

```

Loop through FunctionInstanceList;           // List of function instances
    Loop through (FunctionInstance.Inlist);  // Each input flow
    IF (MaterialList.Find(InFlow) != NULL)   // InFlow is a Material
        M_InList.AddTail(InFlow);          // Collect all material input
    IF (EnergyList.Find(InFlow) != NULL)     // InFlow is an energy
        E_InList.AddTail(InFlow);          // Collect all energy input

    Loop through (FunctionInstance.OutList)  // Each output flow
    Collect all M output to M_OutList and all E output to E_OutList;
    // Using steps similar to collecting input flows

    If ((M_InList.GetCount() == 1) && (M_OutList.GetCount() >= 1)
    // If the one-in-many-out condition satisfies for materials
        Loop through M_OutList using index [i];
        M_InList[0].ChildList.AddTail(M_OutList[i]);

```



```

// Add each material output as a derivative of the single input
    Add "M_InList[0] → M_OutList[i]" to the inference message;

If ((E_InList.GetCount() == 1) && (E_OutList.GetCount() >= 1)
// If the one-in-many-out condition satisfies for energy flows
    Loop through E_OutList using index [i];
    E_InList[0].ChildList.AddTail(E_OutList[i]);

// Add each energy output as a derivative of the single input
    Add "E_InList[0] → E_OutList[i]" to the inference message;

// Before moving to the next function instance, clear off the
temporary storage of input and output lists.
M_InList.RemoveAll();
M_OutList.RemoveAll();
E_InList.RemoveAll();
E_OutList.RemoveAll();

// RemoveAll is a member function of the linked list data structure
that empties the list without deleting it.

```

**Algorithm 4.7: Algorithm for many-in-one-out derivation inference**

```

Loop through FunctionInstanceList;           // List of function instances
    Loop through (FunctionInstance.Inlist);  // Each input flow
    IF (MaterialList.Find(InFlow) != NULL)   // InFlow is a Material
        M_InList.AddTail(InFlow);          // Collect all material input
    IF (EnergyList.Find(InFlow) != NULL)     // InFlow is an energy

```

```

        E_InList.AddTail(InFlow);    // Collect all energy input

Loop through (FunctionInstance.OutList)    // Each output flow
Collect all M output M_OutList and all E output to E_OutList;
// Using steps similar to collecting input flows

If ((M_InList.GetCount >= 1) && (M_OutList.GetCount == 1)
// If the many-in-one-out condition satisfies for materials
    Loop through M_InList using index [i];
    M_InList[i].ChildList.AddTail(M_OutList[0]);
// Add the single material output as a derivative of each input
    Add "M_InList[i] → M_OutList[0]" to the inference message;

If ((E_InList.GetCount >= 1) && (E_OutList.GetCount == 1)
// If the many-in-one-out condition satisfies for energy flows
    Loop through E_InList using index [i];
    E_InList[i].ChildList.AddTail(E_OutList[0]);
// Add the single energy output as a derivative of each input
    Add "E_InList[i] → E_OutList[0]" to the inference message;

// Before moving to the next function instance, clear off the
temporary storage in the input and output lists
M_InList.RemoveAll();
M_OutList.RemoveAll();
E_InList.RemoveAll();
E_OutList.RemoveAll();

```

```
// RemoveAll is a member function of the linked list data structure
that empties the list without deleting it.
```

#### Algorithm 4.8: Algorithm for material transformation without energy

```
Loop through FunctionInstanceList;           // List of function
instances

    // Detect difference between Material flows across every function
    Loop through FunctionInstance.M_InList;   // Each input material
flow
        Loop through M_InFlow.ChildList;     // Each child flow of
InFlow
            IF ((M_InFlow.OutBaggageList.IsEmpty()) &&
                (ChildFlow.InBaggageList.IsEmpty()))
                Echo    "Material    transformation    from"    +
M_InFlow.GivenName + "to" + ChildFlow.GivenName + "without energy
exchange";
```

### 4.2.3 Irreversibility Reasoning Algorithm

#### Algorithm 4.9: Algorithm for qualitative detection of missing residual flow

```
// If there are energy input and output to a function but no residual
energy output, echo the error message.
```

```

Loop through FunctionInstanceList; // List of all function instances

    IF ((FunctionInstance.E_InList.GetCount() > 1) &&
        (FunctionInstance.E_OutList.GetCount() > 1))
    {
        Bool ResidualFound = FALSE;

        Loop through FunctionInstance.E_OutList;

        IF E_OutFlow.IsResidual == TRUE
            ResidualFound = TRUE;

        IF (ResidualFound == FALSE)
            Echo "Residual Energy not found in function: "
            + FunctionInstance.GivenName;
    }

```

#### 4.2.4 Quantitative Reasoning Algorithm (Power Required)

##### Algorithm 4.10: Algorithm for computing power required

```

Loop through FunctionInstanceList; // List of all function instances

    IF ((FunctionInstance.E_InList.GetCount() > 1) &&
        (FunctionInstance.E_OutList.GetCount() > 1))
    {
        Function.PowerReq = Bool ResidualFound = FALSE;

        Function.OutputPower = 0;
    }

```

```

        Loop through FunctionInstance.E_OutList;
        IF E_OutFlow.IsResidual != TRUE

        Function.OutputPower=Function.OutputPower+E_OutFlow.Power;

        FunctionPowerRequired      =Function.OutputPower      /
Function.Effy;
    }

```

**Algorithm 4.11: Algorithm for boundary flow preservation in decomposition**

```

Loop through FunctionInstanceList; // In the low res model
    IF (FunctionInstance.HasDecomposition == FALSE)
        Continue; // If no decomposition, move on to the next
func.
        // HasDecomposition is a Boolean to a function

    IF (FunctionInstance.HasDecomposition == TRUE)
        List <Noun*> InListLowResTemp; // Initiate a temporary list
        Loop through FunctionInstance.M_InList;
            Add M_InFlow to InListLowResTemp; // Add to the
temp list
        Loop through FunctionInstance.E_InList;
            Add E_InFlow to InListLowResTemp; // Add to the
temp list
        // Thus, all pointers of input flows to the low res function

```

```

// are now collected in the temporary list

List <Noun*> InListHighResTemp;// Temporary list at high
res

Loop through HighRes.FlowInstanceList;// All flow instances
// Flow lists in the high resolution model

InListHighResTemp.Add(HighRes.FlowInstance);

// Replicate the input flow list (high res)

// Check if a flow is an input to the decomposed model
IF ((Find(HighRes.FunctionInstanceList, InFlow.HeadNode)) &&
// IF the head is attached to a function in the high res model
(!Find(HighRes.FunctionInstanceList, InFlow.TailNode)))
// but the tail is not connected to a function in the high res
// then the flow is strully an input to the decomposed model
{
    IF (InListLowResTemp.Remove(InFlow) == FALSE)
// Try removing the flow pointer from the low res temp list.
// FALSE means that the input flow is absent in the temp list.
        Echo "Not in low res: " +
InFlow.GivenName;

}

Continue;

// In this manner, all input flows to the high res model that

```

```

// was also found in the low res input list, would be removed
// from the temporary list.  If the decomposition was preserved,
// at this time the temporary list should be empty.

    IF (InListLowResTemp.IsEmpty() == FALSE)
        Echo "Non-preserved flow: " + FunctionInstance.GivenName;

//////////

// Now repeat the process for output flow balancing.
// Exception: Use only non-Loss output flows to check for preservation.
//////////

    List <Noun*> OutListLowResTemp;

    Loop through FunctionInstance.M_OutList;
        IF (M_OutFlow.IsLoss == FALSE)
            Add M_OutFlow to OutListLowResTemp;
    Loop through FunctionInstance.E_OutList;
        IF (E_OutFlow.ISLoss == FALSE)
            Add E_OutFlow to OutListLowResTemp;

    List <Noun*> OutListHighResTemp;

    Loop through HighRes.FlowInstanceList;
    IF (FlowInstance.IsLoss == FALSE)
        OutListHighResTemp.Add(HighRes.FlowInstance);

```

```

    IF ((Find(HighRes.FunctionInstanceList, InFlow.TailNode)) &&
        (!Find(HighRes.FunctionInstanceList, InFlow.HeadNode)))
    {
        IF (OutListLowResTemp.Remove(OutFlow) == FALSE)
            Echo "Not in low res: " +
OutFlow.GivenName;

    }

    Continue;

    IF (OutListLowResTemp.IsEmpty() == FALSE)
        Echo "Non-preserved flow: " + FunctionInstance.GivenName;

```

**Algorithm 4.12: Algorithm for additive inference across decomposition levels**

```

// This algorithm is similar to the one above, with the following
// exceptions:
// 1. It works only on the loss flows
// 2. Instead of checking flow balance across levels, it infers a
// balance.
// 3. It separately accounts for Material and Energy losses, since
// those two subtypes cannot mix.

Loop through FunctionInstanceList; // In the low res model
    IF (FunctionInstance.HasDecomposition == FALSE)

```



```

        Continue; // If no decomposition, move on to the next
func.

        // HasDecomposition is a Boolean to a function

IF (FunctionInstance.HasDecomposition == TRUE)

    List <Noun*> M_OutListLowResTemp;

    Loop through FunctionInstance.M_OutList;

        IF (M_OutFlow.IsLoss == TRUE)

            M_OutListLowResTemp.Add(HighRes.M_OutFlow);

    List <Noun*> M_OutListHighResTemp;

    Loop through HighRes.M_FlowInstanceList;

    IF (HighRes.M_FlowInstance.IsLoss == TRUE)

        M_OutListHighResTemp.Add(HighRes.M_FlowInstance);

    String Message = "";

    Loop through M_OutListLowResTemp;

        Message = Message + M_OutFlow.GivenName + ".EnergyContent +
";

    Message = Message + " = ";

    Loop through M_ OutListHighResTemp;

```

```
Message = Message + M_OutFlow.GivenName + ".EnergyContent +  
";
```

### 4.3 Information Elements Extraction

The algorithms from the previous section are inspected to expose the elements of information used to perform the reasoning tasks computationally. Table 4.16 summarizes these findings against each algorithm. For each information element, its representation component type (entity, relation, or attribute) is captured in the ERA column, and its programming data type and brief description are presented in the last two columns. For each algorithm, only the new information elements identified in addition to previously identified ones are listed. Rows marked as “Nothing new” indicate that no additional element than the ones previously identified are used in that algorithm. The algorithms include keywords, methods, operators, and constants used in common programming languages such as C++ [161], which are not extracted as information element. Examples are (1) keywords such as If, Else, List, String, and Continue, (2) methods such as Find, GetCount, RemoveAll, (3) operators such as “==”, and (4) constants such as NULL, TRUE, and FALSE, which are used in several of the algorithms.

**Table 4.16: Extraction of information elements from the algorithms**

<b>Algorithm Reference</b>	<b>Information Element</b>	<b>ERA Type</b>	<b>Computer Data Type</b>	<b>Description</b>
Algorithm 4.1	Function	Entity	Class Verb	Class from which functions are instantiated
	InList	Attribute	List<Flow*>	Attribute to a function holding the list of flow pointers that are input to a function. Its identification requires identifying the head node of the flow.
	OutList	Attribute	List<Flow*>	Attribute to a function holding the list of flow pointers that are output from a function. Its identification requires identifying the tail node of the flow.
	GivenName	Attribute	String GivenName	Attribute to a to a function, flow, or environment instance in the model, holding the name given to it by user
Algorithm 4.2	Flow	Entity	Class Noun	Class from which flows are instantiated
	TailNode	Relation	Class* Node	Relation between a flow and a

<b>Algorithm Reference</b>	<b>Information Element</b>	<b>ERA Type</b>	<b>Computer Data Type</b>	<b>Description</b>
				node (function of environment) indicating that the node is at the tail of the flow
Algorithm 4.3	HeadNode	Relation	Class* Node	Relation between a flow and a node (function or environment) indicating that the node is at the head of the flow
	Node	Entity	Class Node	Class from which nodes (functions and environments) are instantiated
Algorithm 4.4	ChildList	Relation	List<Flow*>	Relation between a flow and a list of flows, where the flows in the list derived from the given flow.
	Environment	Entity	Class Environment	Class to derive environment instances
Algorithm 4.5	ParentList	Relation	List<Flow*>	Relation between a flow and a list of flows, where the flow is derived into the flows in the list.
Algorithm 4.6	M_InList	Attribute	List<Material*>	Attribute to a function holding the list of material flows input to it

<b>Algorithm Reference</b>	<b>Information Element</b>	<b>ERA Type</b>	<b>Computer Data Type</b>	<b>Description</b>
	E_InList	Attribute	List<Energy*>	Attribute to a function holding the list of energy flows input to it
	M_OutList	Attribute	List<Material*>	Attribute to a function holding the list of material flows output from it
	E_OutList	Attribute	List<Energy*>	Attribute to a function holding the list of energy flows output from it
	M_Flow	Entity	Class Material	Class from which material flows are instantiated
	E_Flow	Entity	Class Energy	Class from which energy flow are instantiated
Algorithm 4.7	Nothing new			
Algorithm 4.8	InBaggage	Relation	List<Flow*>	A relation between two flows, where one flow is carried by the other and the carried flows' head node is same as the carrier flow's head node
	OutBaggage	Relation	List<Flow*>	A relation between two flows, where one flow is carried by the

<b>Algorithm Reference</b>	<b>Information Element</b>	<b>ERA Type</b>	<b>Computer Data Type</b>	<b>Description</b>
				other and the carried flows' tail node is same as the carrier flow's tail node
Algorithm 4.9	IsResidual	Attribute	Boolean	A Boolean attribute to a flow that is set to TRUE if the modeler indicates that an output flow from a function is a loss (residual) flow
Algorithm 4.10	Power	Attribute	Double	A number that indicates the power associated with an energy flow.
	Efficiency	Attribute	Double	A number that indicates the efficiency of a function
Algorithm 4.11	Information elements are not extracted from these two algorithms, since these two algorithms, while detected during the modeling exercise, are out of the scope of this dissertation research, and are reserved for future extensions.			
Algorithm 4.12				

The information elements discovered here are used next to design the formal representation of mechanical functions. In order to organize this design task, the representation is developed in three layers that build successively. The first layer, presented in Chapter 5, is the foundational layer that formalizes basic information elements required for drawing function structure graphs, with formal description of its

entities, relations, attributes, and local grammar to ensure internal consistency of function structure models. In addition, this layer supports qualitative conservation reasoning of the two types identified in this chapter: (1) topologic and (2) derivational. Thus, this first layer is called the **Conservation Layer**.

The second layer extends the first by including the concept of irreversibility and supports reasoning on residual flows at qualitative and quantitative levels. This layer is called the **Irreversibility Layer**.

The third layer is called the **Semantic Layer**. A finite vocabulary of mechanical function verbs is presented that captures the **meaning** of the verb terms through its topology. This layer supports feature-based modeling and semantic reasoning of function structure graphs. In the next chapter, the first layer (Conservation) is developed.

CHAPTER 5. REPRESENTATION LAYER ONE:  
FORMALIZATION OF FUNCTION STRUCTURE FOR CONSERVATION  
REASONING

Based on the reasoning needs and information elements discovered in the previous chapter, the formal representation of mechanical functions is designed in a three-layer structure:

1. **Conservation Layer:** The fundamental entities, relations, attributes, and local grammar rules of function structure modeling are formalized. This layer supports model validation and qualitative reasoning based on the conservation principle, of two types mentioned in Table 4.15: **topologic** and **derivational**.
2. **Irreversibility Layer:** This layer extends the Conservation layer by adding constructs for modeling irreversibility in the vocabulary terms and supports **irreversibility** reasoning.
3. **Semantic Layer:** This layer extends the previous two by adding a vocabulary of physics-based function verbs. It supports feature-based modeling of function structures that also support physics-based reasoning of the previous layers.

This chapter presents the first layer that formalizes the Function Structure representation and supports qualitative conservation reasoning (topologic and derivational). The other layers are subsequently built upon this layer. As defined in



Section 1.2.1 and 1.2.2, a representation is described by its vocabulary and local grammar [30, 31, 110, 162-165], the components of which are:

1. **Vocabulary of entity types:** The vocabulary of unique concept types necessary to construct a model
2. **Vocabulary of relation types:** The vocabulary of unique connection types that can exist between instances of the entity types
3. **Vocabulary of attributes:** The vocabulary of unique parameters used to characterize or specify the entity and relation instances
4. **Local grammar:** The set of rules and constraints that control how instances of entities, relations, and attributes can be combined to construct a model

In the following sections, the entity types, relation types, attributes, and local grammar rules required to formalize the Function Structure representation are identified and rigorously defined. Despite frequent use of function structure models, these graphs have not been formalized as a representation before [48, 115, 116] and do not support consistent formal reasoning [48, 166]. The contribution of this first layer is to that end.

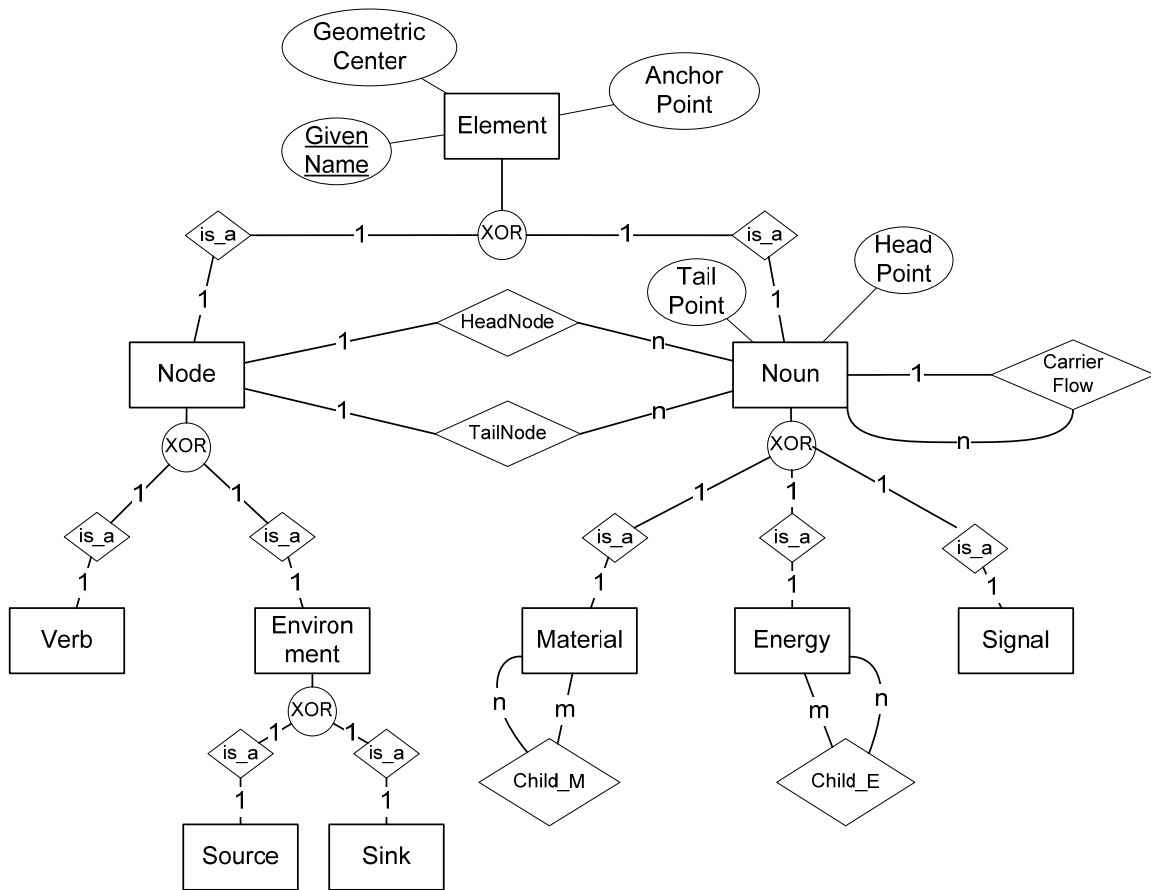
## 5.1 Layer 1 Vocabulary

The vocabularies of entity types, relation types, and attributes are shown in Table 5.1.

**Table 5.1: Layer 1 vocabulary**

		Virtual 1	Virtual 2	Instantiable	Instantiable
<b>Entity Types</b>	::	Element	Node	Verb	
				Environment	Source
					Sink
			Noun	Material	
				Energy	
				Signal	
<b>Taxonomy</b>					
<b>Relation Types</b>	::	{HeadNode, TailNode, CarrierFlow, Child_M, Child_E}			
<b>Attribute Types</b>	::	{GivenName, HeadPoint, TailPoint, GeometricCenter, AnchorPoint}			

The vocabulary of entity types is hierarchical, shown in the taxonomy in Table 5.1, while the vocabularies of relation types and attributes are list based and flat. Not all relation types and attributes are compatible to all entity types. The valid correspondences are shown in in Figure 5.1 using the entity-relation-attribute (ERA) diagram format [162].



**Figure 5.1: Entity-relation-attribute (ERA) model for the Level 1 vocabulary**

The rectangles are entity types, the diamonds are relation types, and the ellipses are the attributes. The labels on the edges of this diagram indicate the one-to-one, one-to-many, or many-to-many correspondence of the relations. For example, the labels 1 and n on the relation type TailNode indicates that a flow can have only one Node instance (function or environment) as its tail node but a Node instance can be the tail node of many flows, which is a one-to-many relationship. The *is\_a* relationship does not appear in the vocabulary (Table 5.1), since it captures the taxonomical and ontological relation between the entity types and is not directly instantiated in a function model. Every other

entity type, relation type, and attribute in this ERA diagram, except Element, Node, and Noun, can be instantiated (Section 5.1.1). The GivenName attribute of every Element instance is its unique identifier. The exclusive disjunction (XOR) symbols on the relations indicate that only one of the many relations connected by the symbol holds for a given instance. For example, a Node instance is either a Verb instance or an Environment instance, but not both.

### **5.1.1 Layer 1 Entity Types**

An entity type is a class that describes a concept, as opposed to an instance of a class, which is a specific occurrence of that concept. The leaf nodes of Figure 5.1 are classes instantiable in models. Classes in the other two columns are virtual superclasses (not instantiable) and are used to organize the vocabulary to minimize redundancy of attributes and relations. For example, the attribute GivenName applies to everything that is an instance of Element and this taxonomy enables declaring this attribute type only once to the Element class, instead of assigning it to five classes in the third column. The methods declared in higher-level classes can also be inherited and overloaded in the subclasses as necessary. In Table 5.2 through Table 5.11, these classes are described using their notional description (natural English), formal description (pseudo code), and graphical description that is used for rendering their instances. Table 5.2 illustrates the entity class Element.

**Table 5.2: Layer 1 entity: Element**

<p><b>Notional Description</b></p>	<p>The most abstract class from which all entity classes are inherited. It holds data members and methods common to all entities, such as the GivenName string and the methods for drawing and editing them.</p>
<p><b>Class Description</b></p>	<pre> Class Element {}           // Base Class { // Data members     String GivenName;      // Name string of the instance     Point GeometricCenter; // Location of the instance on screen     Point[int n] AnchorPoints; // Points where other elements can be attached  // Public virtual methods - declared here, but implemented in subclasses     virtual OnDraw();      // Draw the element     virtual OnSelect();    // Select the element     virtual OnHighlight(); // Highlight the element for selection     virtual OnDelete();    // Delete the element     virtual OnMove();      // Move the element     virtual OnCopy();      // Copy the element     virtual OnSave();      // Save the element     virtual OnLoad();     // Load the element }         </pre>
<p><b>Instance rendering</b></p>	<p>None – not an instantiable class</p>

Table 5.3 illustrates the entity class Node.

**Table 5.3: Layer 1 Entity: Node**

<b>Notional Description</b>	The Node class is used to distinguish the nodes of a function structure (functions and environments) from the flows. This class passes down the data members from Element to Verb and Environment through inheritance and defines one method for computing vertices of the geometric shapes (blocks and hexagons) of functions and environments. The abstract identity of functions and environments as nodes is used by classes such as Noun, which accepts only Node instances as TailNode and HeadNode, thus giving algorithmic flexibility to look for Node instances when querying the terminals of a flow, rather than once looking for functions and then for environments.
<b>Class Description</b>	<pre>Class Node : Element {} // Inherited from Element {     virtual ComputeBlockCoordinates();// Geometric coordinates for rendering }</pre>
<b>Instance rendering</b>	None – not an instantiable class

Table 5.4 illustrates the entity class Noun.

**Table 5.4: Layer 1 entity: Noun**

<p><b>Notional Description</b></p>	<p>Noun is the abstract superclass for the flow classes Material, Energy, and Signal. Inherited from Element, it has the data members and methods that apply to all flow types, irrespective of their subtype.</p>
<p><b>Class Description</b></p>	<pre> Class Noun : Element {}      // Inherited from Element {   // Data members   Node* pTailNode;          // Pointer to tail node (any subtype)   Node* pHeadNode;          // Pointer to head node (any subtype)   Noun* pCarrierFlow;       // Pointer to carrier flow instance   Point TailPoint;          // Location of the tail point on screen   Point HeadPoint           // Location of the head point on screen   // Methods   virtual AssignCarrierFlow; // Implemented in Energy and Signal classes } </pre>
<p><b>Instance rendering</b></p>	<p>None – not an instantiable class</p>

Table 5.5 illustrates the entity class Verb.

**Table 5.5: Layer 1 entity: Verb**

<p><b>Notional Description</b></p>	<p>Verb is the class for deriving function instances. It contains only geometric data that allows constructing the function block. It inherits the Rectangle class, available in most graphics-based development environments such as MFC [167, 168], for this purpose. All methods required for drawing and editing are inherited from the Element class, although overridden specifically within this class.</p>
<p><b>Class Description</b></p>	<pre>Class Verb : Node, Rectangle {} // Inherited from Node and Rectangle {     ComputeBlockCoordinates(); // Computes the rectangle vertices }</pre>
<p><b>Instance rendering</b></p>	<div style="border: 1px solid black; width: 80px; height: 40px; margin: 0 auto; text-align: center; display: flex; flex-direction: column; justify-content: center; align-items: center;"> <span>Function</span> <span>1</span> </div>

Table 5.6 illustrates the entity class Environment.



**Table 5.6: Layer 1 entity: Environment**


<b>Notional Description</b>	Environment is the class for deriving environment instances. It is similar to the Verb class, except that its local implementation of the drawing and editing functions are written to draw hexagonal shapes.
<b>Class Description</b>	<pre>Class Environment : Node {} // Inherited from Node {     ComputeBlockCoordinates(); // Computes the hexagon vertices }</pre>
<b>Instance rendering</b>	

Table 5.7 illustrates the entity class Source.

**Table 5.7: Layer 1 entity: Source**



<b>Notional Description</b>	Source is a class for introducing new flows to the scope of a model. Its instances can only output flows. They cannot input flows.
<b>Class Description</b>	<pre>Class Source : Environment {} // Inherited from Environment {     // None required - the Source behavior is controlled by grammar rules     // applied during the construction of Noun instances - their HeadNode     // cannot point to a Source instance. }</pre>
<b>Instance rendering</b>	

Table 5.8 illustrates the entity class Sink.

**Table 5.8: Layer 1 entity: Sink**

<b>Notional Description</b>	Sink is a subclass of Environment for dismissing flows out of the scope of the model. Its instances can only input flows. They cannot output flows.
<b>Class Description</b>	<pre> Class Sink : Environment {} // Inherited from Environment {     // None required - the Sink behavior is controlled by grammar rules     // applied during the construction of Noun instances - their TailNode     // cannot point to a Sink instance. } </pre>
<b>Instance rendering</b>	

Both Verb and Environment class instances are often queried in the algorithms for their input and output flow lists of the three types (Material, Energy, and Signal). In order to access this topologic connection from the Node, Verb, or Environment classes, lists of flow pointers such Figure 5.2 could be included in the Node superclass. However, a function can have infinitely many input and output flows, while a flow can have only two terminals: head and tail. This character of flows allows easier management of topologic information from the Noun class, as by knowing a **fixed** number of pointers—HeadNode and TailNode—for each flow, the entire graph’s topology can be known. If the lists in Figure 5.2 were used to manage topologic connections, each function would have six lists and each list would contain an **unknown** number of flows, thus calling for dynamic array management challenges.

Including the two Node pointers as well as the lists in Figure 5.2 would cause redundancy of information and loss of normality of the representation. Often, such redundancy is intentionally designed into formal representations. For example, in the boundary representation [32, 33], each edge is accompanied by a co-edge, whose parameter value increases in the opposite direction of the edge and whose vertices are ordered backwards. This redundant co-edge is used to trace an edge in the reverse direction in algorithms such as face stitching [34]. In the proposed function representation, such redundancy is intentionally avoided. When the list of flows attached to a node is needed by an algorithm, the list is composed at runtime by looping through the flows and checking if its head or tail is attached to the node. These temporary lists are stored in a class that represents the model, as discussed in Chapter 6.

List <Element*> pInMaterialList;	// List of input Material flows
List <Element*> pOutMaterialList;	// List of output Material flows
List <Element*> pInEnergyList;	// List of input Energy flows
List <Element*> pOutEnergyList;	// List of output Energy flows
List <Element*> pInSignalList;	// List of input Signal flows
List <Element*> pOutSignalList;	// List of output Signal flows

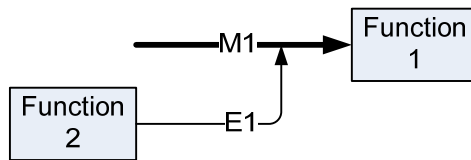
**Figure 5.2: Redundant topologic data elements not captured in any class**

The three classes inherited from Noun have a special method called AssignCarrierFlow that computes the valid carrier types for each flow type. In Material, it sets the pointer pCarrierFlow to NULL. In the Energy and Signal classes, it does two things: (1) it assigns the carrier flow to the instantiated flow and (2) it finds out which end of the instantiated flow is not connected to a node and assigns the corresponding

node of the carrier to the carried flow. For example, in Figure 5.3, queries on the TailNode, HeadNode, and CarrierFlow attributes of M1 and E1 would return the followings. The two bold faced assignments are performed by AssignCarrierFlow.

1. M1.TailNode = NULL, M1.HeadNode = Function1, M1.CarrierFlow = NULL
2. E1.TailNode = Function2, **E1.HeadNode = Function1, E1.CarrierFlow = M1.**

For the graphical depiction of the carrier relationship, the OnDraw() method implemented in the class of E1 ensures that E1.HeadPoint is attached to an AnchorPoint of M1.



**Figure 5.3: Assignment of carrier flow, head node, and tail node based on flow type**

Table 5.9 illustrates the entity class Material.

**Table 5.9: Layer 1 entity: Material**

<p><b>Notional Description</b></p>	<p>Material is the class to instantiate material flows. It inherits all properties from Noun. The only two additional data members are (1) the list of child flows, which must be of the type Material, in order to avoid conversion between material and energy or signal, and (2) the overridden method AssignCarrierFlow.</p>
<p><b>Class Description</b></p>	<pre> Class Material : Noun {           // Inherited from Noun { // Data members     List &lt;Material*&gt; ChildList; // List of derived flows (conservation) // Methods     AssignCarrierFlow();        // Set Noun* pCarrierFlow = NULL                                 // in this local implementation in Material }         </pre>
<p><b>Instance rendering</b></p>	<p style="text-align: center;">—Material—▶</p>

Table 5.10 illustrates the entity class Energy.

**Table 5.10: Layer 1 entity: Energy**


<b>Notional Description</b>	Energy is similar to the Material class, with the difference that its children are all Energy instances and it can be carried by Material instances only.
<b>Class Description</b>	<pre> Class Energy : Noun {}          // Inherited from Noun { // Data Members     List &lt;Energy*&gt; ChildList; // List of derived flows (conservation) //Methods     // The carrier instance is limited to only Material instances     // by a method (local grammar) called by the class constructor.     // This method also assigns the head or tail node of a flow to the     // respective node of the carrier, depending on which end is undefined.     AssignCarrierFlow();      // Carrier flow selection method } </pre>
<b>Instance rendering</b>	

Table 5.11 illustrates the entity class Signal.

**Table 5.11: Layer 1 entity: Signal**

<p><b>Notional Description</b></p>	<p>Energy is similar to the Material class, with the difference that it does not have any children and must be carried by a Material or an Energy instance.</p>
<p><b>Class Description</b></p>	<pre> Class Signal : Noun {Noun* pCarrierME}     // Inherited from Noun {     pTailNode = NULL;        // Signals are not produced by a function     AssignCarrierFlow();     // Carrier flow selection method     // No other declaration is necessary }         </pre>
<p><b>Instance rendering</b></p>	<p style="text-align: center;">-----Signal-----▶</p>

Some behaviors of instances are dictated directly by their definitions, while some are controlled by the local grammar rules. For example, the choice of the class Node for the flow attributes HeadNode and TailNode is built within the definition of the Noun class and prevents anything but a node to be used as a flow terminal. By contrast, the selection of carrier flow type depends on the flow type, since Material does not have a carrier, Energy can be carried only be Material, and Signal must be carried by either Energy or Material. While these carrier relationships could be separately declared in the respective classes, such as “Material\* pCarrierFlow;” within the Energy class, it would remove the ability of accessing the carrier of a flow without knowing its type (Material, Energy, or Signal). For example, when the head of a carrier flow is rerouted from one node to another node, the carried flow’s head node also must be rerouted accordingly.

This editing can be done without knowing the type of the carrier and carried flows, because the CarrierFlow attribute can be accessed from the generic Noun class. Keeping the names of the methods AssignCarrierFlow the same between the two classes and the superclass Noun also adds the flexibility that after rerouting, this method can be called from the Noun class irrespective of their subtype to check for valid carrier relationships.

### 5.1.2 Layer 1 Vocabulary of Relation Types

Table 5.12 describes the relation types in the symbolic layer of the representation.

**Table 5.12: Layer 1 relation types and descriptions**

<b>Relation</b>	<b>In Class</b>	<b>Description</b>
HeadNode (Input)	Noun	HeadNode is a relation between a flow and a node and indicates the node to which the head of the flow is connected. From function modeling point of view, it indicates by which function is the flow used next, or to which environment instance does the flow exit the functional scope of the model. To ensure that the head of a flow can be connected only to a node (function or environment), the attribute data type is set to Node (Table 5.4). For ease of reference, the HeadNode relation is sometimes called the input relation, since a flow is an input to its head node.
TailNode (Output)	Noun	TailNode is a relation between a flow and a node and indicates the node to which the tail of the flow is connected. From



Relation	In Class	Description
		<p>function modeling point of view, it indicates by which function is the flow produced, or from which environment instance does the flow enter the functional scope of the model. To ensure that the head of a flow can be connected only to a node (function of environment), the attribute data type is set to Node (Table 5.4). For ease of reference, the TailNode relation is sometimes called the output relation, since a flow is an output from its tail node.</p>
CarrierFlow	Noun	<p>CarrierFlow is a relation between a flow and another flow. It provides for modeling situations where one flow carries another flow with it. For example, a flow of hot air carries with it kinetic energy and thermal energy.</p>
Child_M	Material	<p>Child_M is a relation between two sets of Material flows and indicates the conservation of the first set in form of the second set. For example, in a chemical reaction, the set of reactants input to the reaction are conserved as the set of products of the reaction.</p>
Child_E	Energy	<p>Child_E is a relation between two sets of Energy flows and indicates the conservation of the first set in form of the second set. For example, when electrical energy enters a motor, it produces many forms of output energy, including mechanical</p>

Relation	In Class	Description
		work, heat, and sound. These output flows are the children of the electrical energy, as the electrical energy is conserved as these flows across a function.

Child\_M and Child\_E relationships exist only between flows that are across a function instance. For ease of reference, the inverse of a Child relation is called the Parent relation.

### 5.1.3 Layer 1 Vocabulary of Attribute Types and their Correspondence

Table 5.13 describes the attributes of the classes, along with their data types.

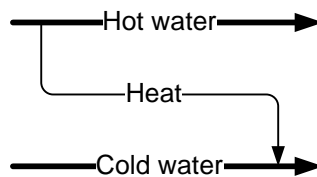
**Table 5.13: Layer 1 attributes and descriptions**

Attribute	In Class	Data Type	Description
GivenName	Element	String	GivenName uniquely identifies every instance in a model. The designer enters a value to this attribute for every instance by giving the instance a name at the time of construction. Methods are written at the model-level classes to avoid duplication of given names.
HeadPoint	Noun	Point	HeadPoint indicates the geometric location of

Attribute	In Class	Data Type	Description
			the head of a flow instance on the screen. It is used to select and edit flow instances and to route the head of a flow to anchor points of the head node.
TailPoint	Noun	Point	TailPoint indicates the geometric location of the tail of a flow instance on the screen. It is used to select and edit flow instances and to route the tail of a flow to anchor points of the tail node.
GeometricCenter	Element	Point	GeometricCenter is the geometric location of the center of an instance on the screen and is used to perform operations such as highlight and select, which are typically followed by edit operations.
AnchorPoints[n]	Element	Point	AnchorPoints[n] is an array of Point data type that holds the possible geometric points where instances can be connected to each other, such as a flow connected to a function or to its carrier flow.

## 5.2 Layer 1 Local Grammar

Except for the trivial cases where a model entirely comprises of only one entity instance, entities must be connected through relations to form a model. Local grammar rules control how entities can be related in a model, by imposing constraints on these relations. Specifically, they prevent connecting instances in a manner that is logically prohibited by the definitions of their own classes, thus ensuring consistency (absence of self-conflict) in the model. These rules are part of the representation, as they are applied at the time of instantiation and control what constructs can and cannot be added to a model, rather than checking a model post-construction for adherence to rules. For example, the definitions of the classes listed above lead to a local grammar rule that prevents a flow to go from one carrier flow to another carrier flow directly. The implication of including this rule in the local grammar is that a properly implemented software based on this representation would make it impossible in the first place to construct the model in Figure 5.4. This behavior is similar to the impossibility of instantiating an edge with three vertices using the boundary representation, as the Edge class requires an Edge instance to be bounded by exactly two Vertex instances.



**Figure 5.4: An internally inconsistent model of a heat exchange function**

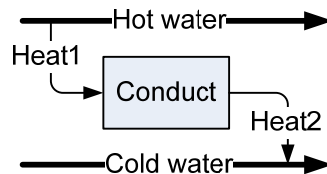
As discussed in Section 1.2.4, in the absence of rigorous formalism of function structure construction, the onus of internal consistency and external validity of models lies on the designer and therefore cannot be ensured. For example, models in the Design Repository [50, 53] show many examples of elements that conflict with their own class definitions (internal inconsistency) and elements that violate laws of conservation (external invalidity), as illustrated in Figure 2.8. By writing and implementing local grammar rules like the one mentioned above, the onus of internal consistency can now be borne by the design of the tool's underlying representation, rather than the designer. However, model validity against external knowledge such as natural laws is not borne by the local grammar and external reasoning algorithms for model validation must be written to ensure that validity. These rules are implemented at the time of reasoning, as opposed to within the representation, and are therefore referred to as global model validation rules.

The significance of local grammar becomes clearer when compared to global model validation rules. Both local and global rules are used to ensure model correctness, although at different levels. The local rules decide whether or not a model is permitted to exist, while the global rules examine if an existing model adheres to some external standard, such as conservation laws. Thus, local grammar ensures consistency (internal) and global rules ensure validity (external). Local grammar rules are applied at instantiation time, while global rules are executed after model construction, to support reasoning [102, 169-171].

A local grammar depends on the collective commitment made in the definitions of all terms in the vocabulary. For example, Figure 5.4 could be interpreted as a model of heat transfer between two water flows at different temperatures, as in a heat exchanger. The modeling construct of adding the tail of the heat flow to a material flow (cold water) is permitted in isolation by the declaration of carrier flows in the Noun class. Similarly, the carrier relation at the head of the flow is also individually permitted, when the other relation is ignored. However, this model is unacceptable as a whole, because it creates ambiguity about the carrier flow of Heat and violates the Noun class definition, since that class defines only one CarrierFlow data member.

Further, this modeling construct contradicts with the definition of function and flow states. State of a flow is defined by the unique combination of its attribute and relation values. Two flows are different (Section 1.2.6) when they are of different types or they are of the same type but at least one attribute or relation value is different between them. Thus, if there are two flows in a model that are of the same type and have the same state, they are indeed the same flow instance and the model is redundant by showing it twice. In the proposed representation, the CarrierFlow relation is included in defining the state of a flow. Thus, heat carried by the hot fluid and that carried by the cold fluid must be two different flow instances, as a difference exists between them in the relation CarrierFlow. Thus, by definition of function, (Section 1.2.7), a function must be involved causing the change of state. In the case of the heat exchanger, this function is the conductive transfer of heat from the hot water to the cold water through a medium such as a pipe wall. Thus, while parts of the model in Figure 5.4 individually agree with

the definition of carrier flows, the model as a whole violates the collective definitions of flow, state, and function. A function is needed to make the model consistent with these definitions, as shown in Figure 5.5.



**Figure 5.5: The consistent construct of the heat exchange function**

Notably, the resulting model still violates known laws of nature, as it does not capture the state change of the water flows, such as the hot water becoming colder and the cold water becoming hotter, and it does not account for heat losses. However, these laws are not part of the definitions of terms **within** the representation, and thus are examples of **external invalidity** rather than **internal inconsistency**. Thus, Figure 5.4 exemplifies internal inconsistency, whereas Figure 5.5 is internally consistent yet externally invalid.

The local grammar for the proposed representation is discussed next. There are three types of modeling constructs that represents relations between the entities: (1) HeadNode and TailNode relations are rendered as the head or tail of an arrow connected to a node, (2) CarrierFlow relations are modeled as a flow head entering or a flow tail leaving the stem of another flow arrow, and (3) Child relation is modeled as derivation dotted lines across a function. These three types are separately presented next, using unary and binary rules.

### 5.2.1 Unary Grammar Rules for Input-Output Relations (HeadNode, TailNode)

Table 5.14 states the unary rules for the input-output connections.

**Table 5.14: Layer 1 grammar rules: Unary input-output relations**

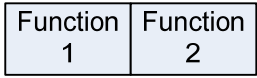

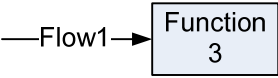

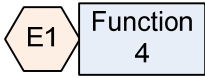

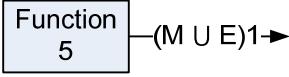

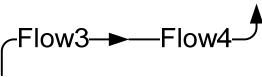

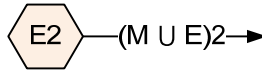

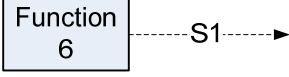

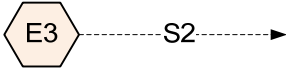

- Rule 1 A flow of any subtype must have exactly one head node (function or environment).
- Rule 2 A flow of subtype Material (M) or Energy (E) must have exactly one tail node.
- Rule 3 A flow of subtype Signal (S) has no tail node.
- Rule 4 Nodes (function, environment) cannot be input or output to each other.
- Rule 5 Flows cannot be input or output to each other.

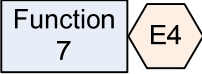

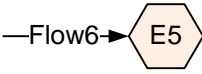

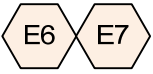

A unary rule determines if a relation between two elements is permitted to exist in a model construct. As illustrated earlier, it is possible to have connections that are acceptable in isolation but not when used in combination with other connections (e.g., the two carrier relations in Figure 5.4). Thus, it is important to distinguish between connections that are prohibited at all from those that are prohibited only when combined with other connections. The benefit is that for connections that are prohibited at the unary level, it is unnecessary to test if they are permitted at the binary level, in combination with some other connection. Table 5.15 shows possible input-output modeling constructs controlled by these rules. Since an input or output relation involves exactly two entities, a total of  $3 \times 3 = 9$  permutations of the form {left entity, right entity} need to be examined, as shown in the table. Further, constructs whose permissions depend on the flow subtypes are explicitly shown. Flows that are labeled as “Flow<n>”



represent any subtype. For each construct, the status is shown with a green check (permitted) or a red cross (prohibited). The rule number that determines the status is shown below the status, using “Ref”.

**Table 5.15: Constructs controlled by the unary input-output rules**

		Left entity		
		Function	Flow	Environment
Right entity	Function	Construct 1   Ref: Rule 4	Construct 2   Ref: Rule 1	Construct 3   Ref: Rule 4
	Flow	Construct 4   Ref: Rule 2	Construct 5   Ref: Rule 5	Construct 6   Ref: Rule 2
		Construct 7   Ref: Rule 3		Construct 8   Ref: Rule 3

		Left entity		
		Function	Flow	Environment
Environment		Construct 9	Construct 10	Construct 11
		  Ref: Rule 4	  Ref: Rule 1	  Ref: Rule 4

These rules are consequences of the class definitions. For example, Rule 1 directly follows from the declaration of one HeadNode and one TailNode pointer in the Noun class. These pointers point to Node instances by the same definition, which explains Rule 5. Similarly, Rule 3 follows from the definition of Signal, which prevents a tail node of a signal flow. In this manner, the rules ensure that constructs inconsistent with the definition of the classes are not permitted in a model. These rules do not impose any constraint on the number of flows attached to a node. These constraints are added later, in Rule 24 and Rule 25 in Table 5.23. Examples of these constructs are found in each of the modeling steps in Section 4.1.

### 5.2.2 Unary Grammar Rules for Carrier-Carried Relations

Table 5.16 describes the unary rules for the carrier-carried relationship.

**Table 5.16: Layer 1 grammar rules: Unary carrier-carried relations**

- Rule 6 A flow can have at most one carrier.
- Rule 7 A flow of type Material or Energy can have null carrier (not a carried flow).
- Rule 8 A flow of type Signal must always have a carrier.

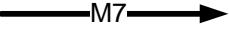

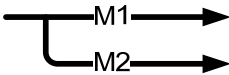

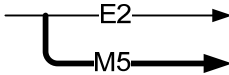

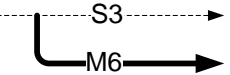

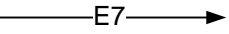

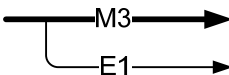

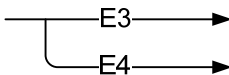

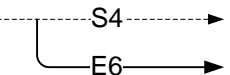

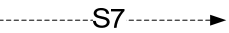

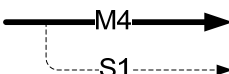

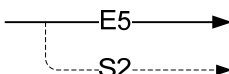

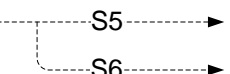

Rule 9 A Material flow can carry one or more flows of subtypes Energy and Signal, but not Material.

Rule 10 An Energy flow can carry one or more flows of subtype Signal, but not of Material or Energy.

Rule 11 A Signal flow cannot carry any flow of any subtype.

Table 5.17 describes the modeling constructs controlled by these rules. Since the carrier-carried relation can only exist between two flows, four possibilities for carrier flows emerge for each of the three types of carried flows, thus producing  $3 \times 4 = 12$  permutations of the form {carried flow, carrier flow}, which are captured in the table. Examples of energy and signal carried by material flows are found in Model State 4.14, where kinetic and thermal energy, as well as the temperature signal, are carried by a hot air flow. An example of signal carried by energy is voltage signal carried by electrical energy flows in various transducers or current signal carried by electrical energy flow in circuit breakers. All other carrier relations are invalid. Signals do not exist in their own identity since they are not physical entities such as material or energy flowing through a device. They are manifested when one or more of the attributes values associated with a carrier flow meets a condition, such as the temperature of Air2 in Model State 4.14 becoming higher than a preset magnitude. Thus, a signal flow always needs to be carried by a flow, whose attributes are decoded or interpreted as signals by a function. For the same reason, signal flows cannot carry any other flows. Rule 6 is used in each cell of the table. A construct that violates Rule 6 is shown in Figure 5.5, where a flow is shown to have two carrier flows. This construct is examined later in Table 5.26.

**Table 5.17: Constructs controlled by the unary carrier-carried rules**

		Carrier flow subtype			
		Null	Material	Energy	Signal
Carried flow subtype	Material	Construct 12   Ref: Rule 6, Rule 7	Construct 13   Ref: Rule 9	Construct 14   Ref: Rule 10	Construct 15   Ref: Rule 11
	Energy	Construct 16   Ref: Rule 6, Rule 7	Construct 17   Ref: Rule 9	Construct 18   Ref: Rule 10	Construct 19   Ref: Rule 11
	Signal	Construct 20   Ref: Rule 6, Rule 8	Construct 21   Ref: Rule 9	Construct 22   Ref: Rule 10	Construct 23   Ref: Rule 11

### 5.2.3 Unary Grammar Rules for Parent-Child Relations

Table 5.18 describes the unary rules for the parent-child relationship.

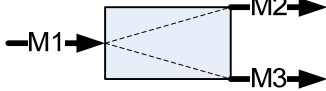



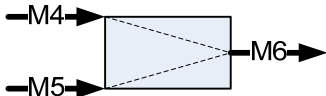



**Table 5.18: Layer 1 grammar: Unary parent-child relations**

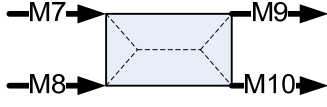



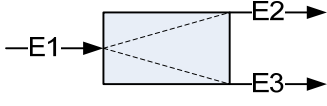


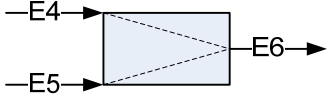

- Rule 12 A flow can be the child of another flow (parent) only if the parent is input to the function of which the child flow is an output.
- Rule 13 A Material flow can have one or more children, all of which must be of type Material.
- Rule 14 A Material flow can have one or more parents, all of which must be of type Material.
- Rule 15 A set of Material flows can be the children of another set of Material flows, where individual derivations are intractable.
- Rule 16 An Energy flow can have one or more children, all of which must be of type Energy.
- Rule 17 An Energy flow can have one or more parents, all of which must be of type Energy.
- Rule 18 A set of Energy flows can be the children of another set of Energy flows, where individual derivations are intractable.
- Rule 19 A Signal flow cannot be the child of any flow.
- Rule 20 A Signal flow cannot have any child flow of any type.


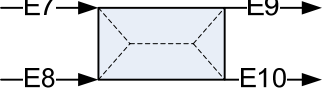











Table 5.19 shows the permutations of model constructs that are controlled by these rules. There are three flow subtypes (Material, Energy, Signal) between which parent-child relations are examined, resulting in  $3 \times 3 = 9$  permutations of the form {parent flow, child flow}. Within each cell, there are three options for cardinality (one in many out, many in one out, and many in many out), resulting into a total  $9 \times 3 = 27$  sub-

permutations (Construct 24 through Construct 50). All constructs in this table satisfy Rule 12. Figures are included only for the consistent constructs.

**Table 5.19: Constructs controlled by the unary parent-child rules**

		Parent flow		
		Material	Energy	Signal
Child flow	Material	Construct 24 One-in-many-out   Ref: Rule 13	Construct 25 One-in-many-out  Ref: Rule 16, Rule 14	Construct 26 One-in-many-out  Ref: Rule 20, Rule 14
		Construct 27 Many-in-one-out   Ref: Rule 14	Construct 28 Many-in-one-out  Ref: Rule 16, Rule 14	Construct 29 Many-in-one-out  Ref: Rule 20, Rule 14

	Parent flow		
	Material	Energy	Signal
	<p>Construct 30</p> <p>Many-in-many-out</p>  <p>Ref: Rule 15</p>	<p>Construct 31</p> <p>Many-in-many-out</p>  <p>Ref: Rule 16, Rule 14</p>	<p>Construct 32</p> <p>Many-in-many-out</p>  <p>Ref: Rule 20, Rule 14</p>
<p>Construct 33</p> <p>One-in-many-out</p>  <p>Ref: Rule 13, Rule 17</p>	<p>Construct 34</p> <p>One-in-many-out</p>  <p>Ref: Rule 16</p>	<p>Construct 35</p> <p>One-in-many-out</p>  <p>Ref: Rule 20, Rule 17</p>	
<p>Construct 36</p> <p>Many-in-one-out</p>  <p>Ref: Rule 13, Rule 17</p>	<p>Construct 37</p> <p>Many-in-one-out</p>  <p>Ref: Rule 17</p>	<p>Construct 38</p> <p>Many-in-one-out</p>  <p>Ref: Rule 20, Rule 17</p>	

		Parent flow		
		Material	Energy	Signal
Signal		Construct 39 Many-in-many-out  Ref: Rule 13, Rule 17	Construct 40 Many-in-many-out   Ref: Rule 18	Construct 41 Many-in-many-out  Ref: Rule 20, Rule 17
		Construct 42 One-in-many-out  Ref: Rule 19, Rule 13	Construct 43 One-in-many-out  Ref: Rule 19, Rule 16	Construct 44 One-in-many-out  Ref: Rule 19, Rule 20
		Construct 45 Many-in-one-out  Ref: Rule 19, Rule 13	Construct 46 Many-in-one-out  Ref: Rule 19, Rule 16	Construct 47 Many-in-one-out  Ref: Rule 19, Rule 20
		Construct 48 Many-in-many-out  Ref: Rule 19, Rule 13	Construct 49 Many-in-many-out  Ref: Rule 19, Rule 16	Construct 50 Many-in-many-out  Ref: Rule 19, Rule 20

In summary, material is conserved as material and energy is conserved as energy only. Although conversion from material to energy is possible physically, such as in



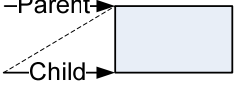

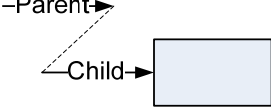
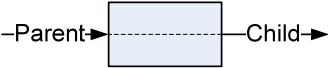
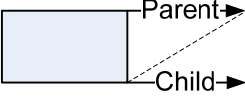
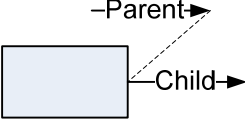
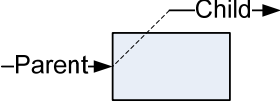
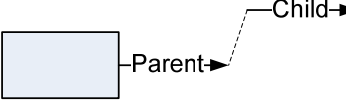
nuclear reactions, Construct 33, Construct 36, or Construct 39 are not allowed in this representation, as allowing them would make it difficult for the reasoning algorithms to distinguish nuclear reactions that convert mass to energy from models that are subject to the separate laws of mass and energy conservation and yet show one of these three constructs due to oversight or wrong ideation. It is anticipated that devices and principles that are subject to the separate conservation laws of mass and energy will be more frequently encountered in mechanical design projects than nuclear reactions. By design, this decision reduces the representation's coverage over nuclear phenomena. However, in the design of a nuclear power plant, this compromise would only eliminate the ability to model the nuclear fuel and reaction principles, while retaining coverage over the other principles and devices. Even in a nuclear plant, most machinery such as boilers, turbines, and heat exchangers are subject to the separate conservation laws and can still be modeled.

The need for conservation does not apply to signals, as they are not entities that exist on their own. Signals are realized as the attribute values of a material or energy flow satisfy a condition that is previously agreed upon between the sender and the recipient as a message. For example, in a thermostat, a signal to actuate an electrical switch may be manifested by the temperature attribute of a material flow satisfying a condition. The material flow itself is not the signal, since it could exist even if its temperature was not measured. Temperature is not a signal, as it is not a flow, rather, an attribute of a flow. The phenomenon of the temperature attribute satisfying the condition is what is interpreted by another device as the signal. Thus, the existence of a signal flow

requires the presence of the interpreting device or function. Nothing is **consumed** to produce this signal, nor is the signal consumed to produce anything. Thus, signals are not conserved.

The 27 cells in Table 5.19 describe topological configurations that satisfy Rule 12. Ideally, there are 27 similar permutations of type and cardinality for the topology where the parent flow set is not input to the function that produces the children set and 27 more for when the children are not output from the function to which the parent set is input. In between these two, there are many more possibilities depending on the exact count of flows in the parent and child sets for some flows in a set being input or output to a function across which conservation is examined. Table 5.20 summarizes the status outcomes of these constructs using only nine simplified permutations. There are three ways by which each of the parent and the child flow can be related to a function through the input-output relation: (1) the flow is input to the function, (2) the flow is output from the function, and (3) the flow is not connected to the function. Thus, there are a total of  $3 \times 3 = 9$  permutations to be considered. For each case, the cell illustrates the status of the construct as per Rule 12. For simplicity, the terms Parent and Child include all flow subtypes. The status applies to all cardinality options, although the illustrations use one-in-one-out. With these simplifications, these nine permutations cover all possibilities explained above.

**Table 5.20: Constructs controlled by the unary parent-child rules (Rule 12)**

		Parent			
		Input	Output	None	
Child	Input	Construct 51  ❌ Ref: Rule 12	Construct 52  ❌ Ref: Rule 12	Construct 53  ❌ Ref: Rule 12	
		Output	Construct 54  ✔️ Ref: Rule 12	Construct 55  ❌ Ref: Rule 12	Construct 56  ❌ Ref: Rule 12
			None	Construct 57  ❌ Ref: Rule 12	Construct 58  ❌ Ref: Rule 12

#### 5.2.4 Binary Grammar Rules for Input-Output Relations

The binary grammar rules examine if two relations can coexist in a construct. For input-output relations, these rules examine if the two ends of a flow can be connected to a

certain permutation of nodes (functions and environments). Table 5.21 states the binary input-output rules.

**Table 5.21: Layer 1 grammar: Binary input-output relations for flows**

- Rule 21 A flow of subtype Material or Energy can be output from a node and input to a different node.
- Rule 22 A flow of any type cannot be output from and input to the same node.
- Rule 23 A flow of any type cannot have environment instances as both HeadNode and TailNode.

For a binary construct to be permitted, all unary constructs used within it must be permitted. However, all binary constructs composed of permitted unary constructs are not necessarily permitted. For example, a binary construct where a Signal flow output from a function block is input to another function block is prohibited as it contains a prohibited unary construct where a Signal flow is produced by a function (see Construct 7 in Table 5.15). However, although a binary construct where a flow is output from and input to the same function is permitted by combining the unary Construct 2 and Construct 4, it is not permitted as a binary construct. Those relations are not allowed to coexist, as they imply circular dependency (see Table 5.22). Thus, the legality of the constituent unary constructs is a necessary but not sufficient condition for the legality of a binary construct. Table 5.22 illustrates only those binary constructs that are composed of the permitted unary constructs. For each row, the unary constructs used to compose the binary construct are mentioned in the first two columns, while the rule used to determine

the status is mentioned in the Ref. column. The flows in each case indicate either Material or Energy subtype, as Signal flows are not permitted to have tail nodes.

**Table 5.22: Constructs controlled by the binary input-output rules**

Tail node	Head node	Binary Construct	Ref.	Rationale
Function Construct 4	Function (other) Construct 2	<p>Construct 60</p>	Rule 21	The flow is produced by an action of the device and used by another.
Function Construct 4	Environment Construct 10	<p>Construct 61</p>	Rule 21	The flow is produced by an action of the device and further actions on it are not in the model scope. This construct is analogous to the verb Export in the Functional Basis.
Environment Construct 6	Function Construct 2	<p>Construct 62</p>	Rule 21	The flow is used by an action of the device, but the previous action that produced it is not in scope. This construct is analogous to the verb Import in the Functional Basis.

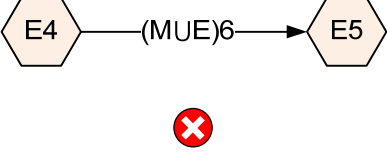
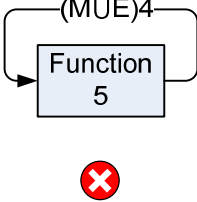
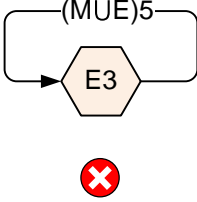
Tail node	Head node	Binary Construct	Ref.	Rationale
Environment Construct 6	Environment (other) Construct 10	<p>Construct 63</p> 	Rule 23	Environments are boundary entities of a model. The flow thus connected never entered the model, and is thus redundant.
Function Construct 4	Function (same) Construct 2	<p>Construct 64</p> 	Rule 22	Since each flow arrow represents one state of the flow, the function outputs the flow in the same state as input, and is thus redundant.
Environment Construct 6	Environment (same) Construct 10	<p>Construct 65</p> 	Rule 22	Environments are boundary entities of a model. The flow thus connected never entered the model, and is thus redundant.

Table 5.22 only includes constructs of the form {node, flow, node}. Two more constructs—of the form {flow, node, flow}—are possible for the two subtypes of nodes. Table 5.23 describes the binary rules for these constructs, where two or more input-output relations are considered for a node.

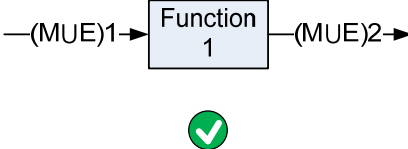
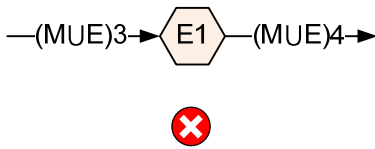
**Table 5.23: Layer 1 grammar: Binary input-output relations for nodes**

Rule 24 A function has no restriction on the number and type of input and output flows attached to it, as long as the flow constraints are satisfied.

Rule 25 An environment instance can be either source or a sink, but not both.

Table 5.24 illustrates the constructs controlled by these two rules. As before, the flows include Material and Energy instances, but not signals, as they cannot have tail nodes. The rules apply for all options of cardinality, although the figures are drawn using the one-in-one-out cardinality.

**Table 5.24: Constructs controlled by the binary input-output rules**

Tail node	Head node	Binary construct	Ref.	Rationale
Function Construct 4	Function (same) Construct 2	<p>Construct 66</p> 	Rule 24	This construct can occur when a function receives a flow and produces its derivative.
Environment Construct 6	Environment (same) Construct 10	<p>Construct 67</p> 	Rule 25	See rationale below.

Environments represent boundaries of the modeling scope. By choice, it is decided that in a model, each environment is either a source of flows or a sink of flows. A flow through an environment instance can exist only when two models are joined or

composed. Here, a flow produced by one model (Flow1) is used by another model as input (Flow2). Grammar rules could be written to allow junctions of this type temporarily and to check if the two flows are at the same state, in which case the operation could be finished by destroying the environment instance and connecting the two flows. However, this operation is out of the scope of this research.

### **5.2.5 Binary Grammar Rules for Carrier-Carried Relations**

By definition of the class Noun (Table 5.4), each flow can have at most one carrier and thus, the carrier relation is inherently unary. Further, the unary constructs in Table 5.17 illustrate that only three types of carrier-carried relations are valid: (1) material carrying energy, (2) material carrying signal, and (3) energy carrying signal. However, the definition of Noun or the unary rules do not describe how carried flows can be added to or extracted from their carrier flows. The binary rules impose those constraints.

Table 5.25

#### **Table 5.25: Layer 1 grammar: Binary carrier-carried relations**

- Rule 26 A carried Energy flow can be added to a carrier Material flow only by the node that outputs the carrier Material flow.
- Rule 27 A carried Energy flow must be extracted from a carrier Material flow as input by the node that inputs the carrier Material flow.
- Rule 28 A signal flow is never added to a carrier flow.
- Rule 29 A carried signal flow can be extracted from its carrier by the same or a different node that inputs the carrier flow.



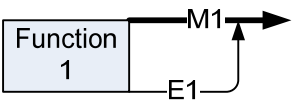
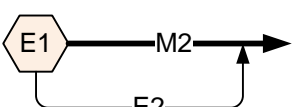
Rule 30 Addition and extraction of a carried flow to and from a carrier flow cannot be accomplished without an intervening function.

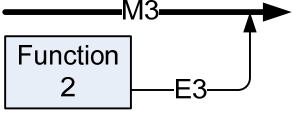
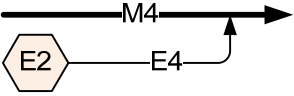
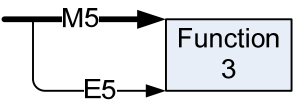

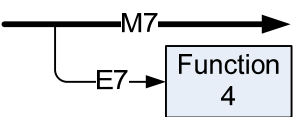
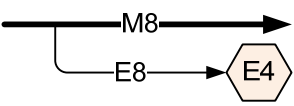
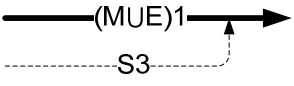
Rule 31 A node can input a carried flow only if its carrier flow is also an input to the node.

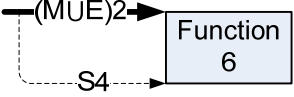
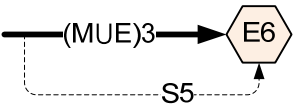
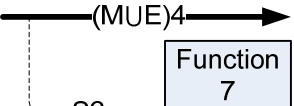
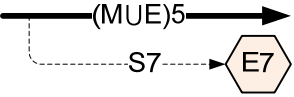
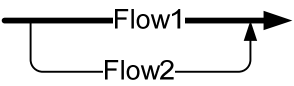
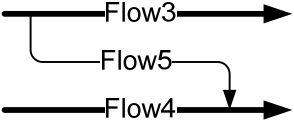
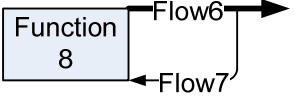
Rule 32 A node can output a carried flow only if its carrier flow is also output by the node.

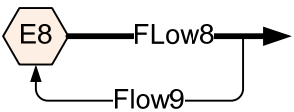
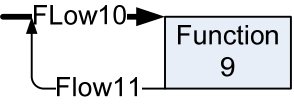
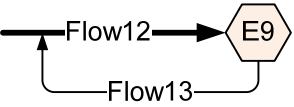
Table 5.26 illustrates the modeling constructs controlled by these rules. In the last cluster in this table, rules are mentioned that apply to all subtypes of carrier and carried flows. These rules depend on the head and tail connections of both carrier and carried flows, as mentioned in columns 2 through 5. The rationale for each rule is provided following the table, using reference to construct numbers.

**Table 5.26: Constructs controlled by the binary carrier-carried rules**

Relation	Carried flow tail	Carried flow head	Carrier flow tail	Carrier flow head	Binary construct	St.	Ref.
Node	Carrier	Node (same)	Carrier	Carrier	Construct 68 	✓	Rule 26
					Construct 69 	✓	Rule 26

Material carrying energy						
	Node	Carrier	$\neg$ Node (same)	<p>Construct 70</p> 	✘	<p>Rule 26 Rule 32</p>
				<p>Construct 71</p> 	✘	<p>Rule 26 Rule 32</p>
	Carrier	Node	Node (same)	<p>Construct 72</p> 	✔	<p>Rule 27</p>
				<p>Construct 73</p> 	✔	<p>Rule 27</p>
	Carrier	Node	$\neg$ Node (same)	<p>Construct 74</p> 	✘	<p>Rule 27 Rule 31</p>
				<p>Construct 75</p> 	✘	<p>Rule 27 Rule 31</p>
	Carrier			<p>Construct 76</p> 	✘	<p>Rule 28</p>

Material OR energy carrying signal				Carrier	Node			Construct 77 	✓	Rule 29
				Carrier	Node			Construct 78 	✓	Rule 29
				Carrier	Node			Construct 79 	✓	Rule 29
				Carrier	Node		¬ Node (same)	Construct 80 	✓	Rule 29
				Carrier	Carrier (same)			Construct 81 	✗	Rule 30
				Carrier	Carrier (different)			Construct 82 	✗	Rule 30
Any carrying any				Carrier	Carrier's tail node	Function		Construct 83 	✗	Rule 31

	Carrier	Carrier's tail node	Environment		Construct 84 	⊗	Rule 31
	Carrier's head node	Carrier		Function	Construct 85 	⊗	Rule 32
	Carrier's head node	Carrier		Environment	Construct 86 	⊗	Rule 32

Rationale behind the Binary Carrier-Carried Modeling Constructs

Construct 68. This construct represents the production of an energized flow. The function outputs both the M1 and E1, while E1 leaves riding on M1. An example is a flow of moving air (M1), which carries kinetic energy (E1).

Construct 69. This construct is redundant, but not prohibited. It implies introduction of an energized flow to a system. The energy flow is redundant, since it can be modeled as input to the first function that uses it, as shown in row 5. This construct is similar to the verb Import in the Functional Basis.

Construct 70. The output of energy E3 is not prohibited. Its addition to the material flow M3 is prohibited, since it does not capture for the state change of the material due to this energy added. In this representation, each

arrow is a specific flow state and the state cannot change from the tail of the arrow to its head. For example, if E3 was heat and M3 was air, it is not clear if M3 is the colder air or the hotter air. By definition of function, a function must be present to depict the **addition** of heat to air and the change of state, using two flows of air across the function. Function2 only shows the production of heat, not the addition to air. The example in row 1 has a similar construct, but the depiction that M1 and E1 are produced by the same function, coupled with the fact that E1 is carried by M1, indicates that the function not only produces the energy, but also adds it to the material flow. Thus, there are two fundamental actions lumped in Function1.

Construct 71. This construct is prohibited because of similar reasons explained above, applied to environment instances instead of functions.

Construct 72. This construct is the input-side counterpart of row 1, and implies that the function inputs the material flow M5 to extract its energy E5 to perform its actions.

Construct 73. This construct is the export counterpart of row 2. The depiction of E6 is not incorrect, but redundant.

Construct 74. This construct is prohibited because of the reasons explained in row 3. The change of state of M7 must be captured in a separate function.

Construct 75. This construct is prohibited for the same reasons explained above, applied to environments.

Construct 76. As explained earlier, signals are manifested when attributes of a material or energy flow meets a previously agreed condition. This condition is tested by the function that uses the signal, not by the function that produces the carrier flow of the signal. For example, in the air heater function structure (Model State 4.14), the temperature attribute of Air2 is interpreted as a signal by the function Conduct EE. The Transfer Air function, which produces Air2, is responsible for causing the temperature attribute to change, but it does not influence or depend upon the fact that this temperature attribute is used by another function as a signal. For example, in an open-loop system, the Transfer Air function would still produce Air2 at certain temperature and would not change in terms of the types and count of flows. The information transferred to Conduct EE is held in Air2, not in the function that produces it. In this sense, a signal flow is never added by anything to a carrier. A signal is only interpreted from the attributes of a carrier.

Construct 77. It is possible and acceptable that the signal is used by a function that also inputs the flow whose attribute is used as a signal. For example, a circuit breaker uses the current attribute of an electrical energy flow that it also inputs.

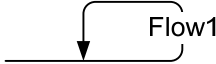
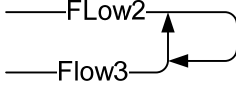
Construct 78. Similar argument as above, applied to environments.

- Construct 79. As explained with the air heater example, it is possible and acceptable that an attribute of a flow is interpreted as signals by a function other than the function that uses that flow. While this other function (Function6) uses the information carried by the flow (MUE)4, it does not perform a transformative action on the carrier flow, and thus should not input it.
- Construct 80. The same argument as above, applied to environments.
- Construct 81. This construct is a redundant depiction of a carrier flow.
- Construct 82. This construct violates the definition of the term function, as illustrated in the beginning of Section 5.2, in context of Figure 5.4 and Figure 5.5.
- Construct 83. By definition of function, an output flow at its state is the outcome of a transformative action and is produced **after** the action is performed. Thus, a flow carried by that output flow cannot be used by the function, as that would imply circular dependency.
- Construct 84. The same argument applies, for environment instances.
- Construct 85. In symmetry to the above argument, an input flow is required to perform the function and must therefore exist at its state **before** the function is performed. Thus, an output of the function cannot be carried by it, since the output is produced **after** the function execution.
- Construct 86. Same argument as above, applies to environment instances.

### 5.2.6 Special Grammar Rules for Layer 1

In addition to the rules mentioned above, a special rule is necessary to prevent circular dependency of carrier flows (Table 5.27).

**Table 5.27: Layer 1 grammar: Special rule**

Rule description	Graphical construct	St.
Rule 33: A set of flows cannot form a circular chain of carrier-carried relations.	Construct 87 	✘
		✘

The carrier relation is predicated upon the understanding that the specific carried flow instance cannot exist without the carrier. For example, kinetic energy of a moving body exists on the condition that the body itself exists. The carrier is capable of existing without the carried flow. For example, the body could exist at rest and have no kinetic energy. Then, if Flow2 is capable of carrying Flow3, it can independently exist too. The same argument applies to Flow3 and thus raises the question why any of the two flows should need a carrier. The same argument can be extended to larger loops of circular dependency of carrier relations. Notably, while this construct can be rejected based on this circular dependency argument, in this representation, this construct is already prevented by the hierarchy of carrier flow subtypes in Table 5.17. For example, since material can carry energy, but the reverse is not true, Flow2 and Flow3 cannot be the



carrier of each other, no matter what their subtypes are. Thus, the circular dependency clause is redundant for this representation.

In conclusion, this chapter formalizes the Function Structure representation, which constitutes the first layer of the proposed representation. Specifically, entities, relations, attributes, and local grammar rules to ensure internal consistency of models are formally defined. The next necessary step is to validate that this proposed layer is in fact capable of constructing function structures supported by existing research and of supporting the claimed reasoning types: topologic and derivational conservation, which is presented next.

CHAPTER 6. IMPLEMENTATION AND VALIDATION:  
MODELING AND REASONING DEMONSTRATION WITH LAYER ONE

Validation of a representation includes demonstrating internal consistency and external validity [31]. Consistency requires that the declarations within the representation do not lead to self-contradiction through logical inference and is demonstrated from three directions (Section 6.1):

1. **Exhaustiveness of Local Grammar** (Section 6.1.1): Logically examinations are used to show that all constructs possibly considered for inclusion in a function structure are identified in the grammar and for each construct, there is a rule to test if the construct is permitted or prohibited.
2. **Consistency of Local Grammar** (Section 6.1.2): Logical examinations are used to show that the rules available in the local grammar do not contradict mutually.
3. **Consistency of Vocabulary** (Section 6.1.3): The class definitions are committed to an ontology using Protégé-OWL<sup>11</sup> and checked using the Protégé logical consistency checker.

---

<sup>11</sup> <http://protege.stanford.edu/overview/protege-owl.html>, accessed on August 16, 2011

Validity against external knowledge requires that statements made using the representation that are known to be correct by that body of knowledge do not lead to inferences that are not correct by that body of knowledge. In this dissertation, the external bodies of knowledge against which validity is sought are (1) the laws of conservation and (2) irreversibility. In this section, validity against conservation is illustrated in three steps (Section 6.1.3).

1. **Software Implementation** (Section 6.2.1): The representation is implemented in software.
2. **Reasoning Demonstration** (Section 6.2.2): The software tool is used to construct models of low complexity which are then used to perform qualitative conservation reasoning of topologic and derivational types, showing that the inferences are in agreement with the conservation principle.
3. **Product-Level Modeling Demonstration** (Section 6.2.3): The software is used to construct the air heater function model from Model State 4.14 to illustrate that the representation can support larger models and perform reasoning on them, thus illustrating scalability at a basic level.

Through this illustration, it is also shown that while the first layer is valid against conservation, it is not valid against the irreversibility principle. This gap motivates the design of the subsequent layers. In the next section, the internal consistency of the first layer is demonstrated.

## 6.1 Demonstration of Internal Consistency

One way of testing a representation for consistency is to implement it in software and then to try to construct models that are allowed by the tool yet are illogical based on the definitions of the terms. In this approach, lack of consistency can be established by finding at least one instance where a model construct violates the definition of the elements used in it. However, the inability to produce such a violation does not conclusively provide its consistency, although a high number of failed attempts to simulate inconsistency may give reasonable confidence toward it. The only way to **prove** consistency is to examine the definitions and grammar rules exhaustively and show that none of their implications leads to self-contradiction. However, this approach is practically infeasible, since it requires testing many model constructs, as explained next.

The total number of model elements in the first layer includes ten classes, five relations, five attributes, and 33 grammar rules (Chapter 5). For simplicity, the attributes and relations are not considered in this discussion. Since a definition can be inconsistent within itself or contradict with other definitions, consistency checking for the ten class definitions requires checking  $(2^{10} - 1) = 1023$  subsets. The negative one indicates that the empty set, consisting of no class at all, needs not be tested. Similarly, the grammar rules, which could be inconsistent within oneself or between two or more rules in combination, form a total number of test cases of  $(2^{33} - 1)$ , a ten-digit number. Further, a subset of classes that is consistent as a set of definitions can still be inconsistent when combined with one of the grammar rule sets, which requires checking every possible combination between class sets and rule sets – a total option space of  $(2^{10} - 1) \times (2^{33} - 1)$ ,

a thirteen-digit number. Thus, the total number of combinations of definitions and grammar rules that need to be checked is:  $(2^{10} - 1) + (2^{33} - 1) + (2^{10} - 1) \times (2^{33} - 1)$ , a number larger than  $8.796 \times (10)^{12}$ . To provide some perspective, if each evaluation takes only one second—a highly ambitious estimate—the total time required to finish these evaluations is over 278922 years.

A **proof** of consistency being infeasible, this section illustrates with combinatorial arguments how consistency is built in the design of the representation, especially in the grammar rules. **First**, it is shown that all constructs and rules necessary for function structure construction are available in the grammar (exhaustiveness). **Second**, it is shown that the rules available are mutually consistent (consistency). If all necessary rules are available and they are consistent, the grammar as a whole is claimed to be consistent. **Finally**, an ontological examination reveals that the class definitions are internally consistent, thus establishing that the entire representation is internally consistent.

### 6.1.1 Logical Examination of Exhaustiveness of Local Grammar

The exhaustiveness of the grammar is examined in two steps. **First**, it is shown that all rule types necessary for model construction are captured in the grammar, where each table in Section 5.2 represents a rule type. **Second**, it is argued that within each rule type, all possible combinations of the constructs are considered, where each row a table is one combinatorial possibility. Together, these two claims imply that while constructing a function structure, a modeler can always find a grammar rule to check if a possible construct is prohibited.

### Exhaustiveness of Rule Types

The local grammar provides all necessary rule types to control model construction. The ERA model for the first layer (Figure 5.1) reveals only three basic relation types: (1) input-output, (2) carrier-carried, and (3) parent-child. Since grammar rules control the permission or prohibition of relations instances, rules must exist for each relation type for exhaustiveness. To this end, grammar rules exist for each relation: Table 5.14, Table 5.21, and Table 5.23 control the input-output relation, Table 5.16, Table 5.25, and Table 5.27 control the carrier-carried relation, and Table 5.18 and Table 5.20 control the parent-child relation. Further, since a flow has exactly two ends (head and tail) and a node has exactly two ends (input and output), only two levels of connections needs to be examined for the input-output relation: unary and binary. The unary rules examine if a relation can exist at all between two entities. The benefit of separately stating the unary rules is that for relations that are prohibited at the unary level, it is unnecessary to further examine if they can coexist with other relations. Once a relation can exist at one end of a node or flow, the binary rules determine if a relation can exist at the other, open end of the node or flow. For the input-output relation, Table 5.14 captures the unary rules, Table 5.21 captures the binary rules for flows, and Table 5.23 describes the binary rules for the nodes, thus giving exhaustive coverage on input-output rule types.

Since a flow can have only one carrier, the carrier-carried relation is inherently unary, the rules for which are captured in Table 5.16. However, rules must exist to control two actions: the addition and extraction of a flow to or from its carrier. These rules are the binary carrier-carried rules captured in Table 5.25, and Table 5.27. For the

parent-child relation, although a flow can have multiple parents of children, only one instance of the relation exists between the parent set and the child set. Thus, this relation is inherently unary, captured in Table 5.18 and Table 5.20. In summary, grammar rule types exist for all relations and all varieties (unary, binary). Thus, the set of grammar rule types is exhaustive.

#### Exhaustiveness of Constructs and Rules within Each Rule Type

While individual rule types exist for all relations, it can be shown by examining the rule tables and the constructs in Section 5.2 that within individual rule types, enough rules exist to control modeling. For example, since an input or an output relation involves exactly two entities and there are three major types of entities—functions, flows, and environments—it is sufficient to examine  $3^2 = 9$  constructs of the form {first entity, second entity} to test all constructs of the input-output unary rules. Table 5.15 shows that the unary rules in Table 5.14 do provide a decision for each of these possibilities. To be sure, if the three subtypes of flows are separately counted, the number of entities increases to five: function, environment, material, energy, and signal. In that case, a total  $5^2 = 25$  permutations need to be tested. While Table 5.15 does not have 25 cells, it does cover those 25 permutations by lumping the three flow types into Flow<n> whenever possible. If the status of a construct depends on the flow subtype, those constructs are explicitly shown, e.g., in the second row of Table 5.15. Thus, rules exist for all possible unary constructs and therefore, the unary rules for input-output relations (Table 5.14) are exhaustive.

The binary input-output rules are written only for the constructs that are valid at the unary level. Since there are only two permitted constructs for a flow output from a node and two more for a flow input to a node (Table 5.15), a total of  $2 \times 2 = 4$  combinations of the form {node, flow, node} are to be examined, where the head and the tail nodes are different instances. In addition, two more possibilities arise where the nodes are the same. Thus, only six possible binary constructs need to be examined for the input-output relation, all of which are captured in Table 5.22. Similarly, for binary constructs of the form {flow, node, flow}, the four permitted unary constructs in Table 5.15 can be combined in only two sets that has a node in the middle. These two sets are examined in Table 5.24, providing evidence that the rules in Table 5.23 are exhaustive. Thus, both unary and binary rules for the input-output relation are exhaustive.

In this manner, the exhaustiveness of the carrier-carried rules and the parent-child rules can be verified by examining the possible combinatory constructs captured in their respective tables. This exercise is omitted here to avoid repetition—the combinations are explained for each rule type in the paragraph preceding the table of illustrations in Chapter 5. For example, the unary carrier-carried rules must support  $3 \times 4 = 12$  constructs (Table 5.17), as there are three subtypes of carried flows and four possibilities of carrier flow subtypes. Similarly, the parent-child rules (unary only) must support  $3 \times 3 \times 3 = 27$  constructs (Table 5.19), as there are three subtypes of the parent flow, three subtypes of the child flow, and three options for cardinality (one-in-many-out, many-in-one-out, and many-in-many-out). By examining these tables, it is seen that the tables



capture all possible combinatory constructs and the rules provide for a decision for each construct. **Thus, rules are exhaustively written for all possible modeling constructs.**

### 6.1.2 Logical Examination of Consistency of Local Grammar

The illustrations in the previous two subsections establish that there is a rule to determine if any possible modeling construct is prohibited. In this subsection, it is shown that within a rule type, the rules are consistent. Since each rule determines values for a particular parameter, such as the number of head nodes of a flow, a rule can be thought of as a mathematical function (*math\_func*) between its argument and outcome. The keyword *math\_func* is used to distinguish mathematical functions from mechanical functions. Three concepts—domain, range, and *math\_func*—are explained here. (1) Domain of a rule is the set of argument entities for which the rule is defined. For Rule 1, the domain is {Flow} since the rule can be applied to any flow entity to compute its number of head nodes. (2) *Math\_func* of a rule is the parameter that the rule returns. Rule 1 determines the number of head nodes permitted for the flow argument and thus, *math\_func* = Number\_of\_Head\_Nodes. (3) Range of a rule is the set of values that the rule can produce. The range of Rule 1 is {1}. In order to have the possibility of self-conflict, two rules must have the same *math\_func* and intersecting domains. Otherwise, either they compute different parameters for the same domain, compute same parameters for different domains, or are totally unrelated, each of which eliminates the possibility of mutual conflict. A sufficient condition for conflict is for two rules to meet the necessary conditions and in addition have disjoint ranges. Thus, one way to demonstrate internal

consistency is to disprove the necessary condition by showing that there are no set of rules with the same `math_func` and intersecting domains.

Table 6.1 describes these three properties for the unary input-output rules, first introduced in Table 5.14. Since the rules are written in natural English for easier interpretation and since some of these rules include negations, the domain and range may be sometimes difficult to identify. For example, Rule 4 is defined over the domain  $\{\text{Node}\}$ , while its range is  $\{\neg \text{Node}\}$ .

**Table 6.1: Domain, `math_func`, and range for unary input-output rules**

<b>Rule #</b>	<b>Domain</b>	<b><code>math_func</code></b>	<b>Range</b>
Rule 1	$\{\text{Flow}\}$	<code>Number_of_Head_Nodes</code>	$\{1\}$
Rule 2	$\{\text{Material} \cup \text{Energy}\} \subset \{\text{Flow}\}$	<code>Number_of_Tail_Nodes</code>	$\{1\}$
Rule 3	$\{\text{Signal}\} \subset \{\text{Flow}\}$	<code>Number_of_Tail_Nodes</code>	$\{0\}$
Rule 4	$\{\text{Function} \cup \text{Env}\} \subseteq \text{Node}$	<code>Permitted_Input_Types</code>	$\{\neg \text{Node}\}$
Rule 5	$\{\text{Flow}\}$	<code>Permitted_Input_Types</code>	$\{\neg \text{Flow}\}$

A scrutiny of Table 6.1 reveals that there are no two rules for which the domain and `math_func` are the same. For example, Rule 1 and Rule 4 have the same domain but different `math_func`, while Rule 2 and Rule 3 have the same `math_func` but their domains are non-intersecting, since Flow is the disjoint union of Material, Energy, and Signal. Thus, the unary input-output rules are mutually consistent. The method of examination illustrated above can be repeated for all rules in the local grammar. Since there are 33

rules, there are  $33 \times 32 = 1056$  rule pairs to examine. While that examination is not included, the domain, math\_func, and range of all rules are presented in Table 6.2.

**Table 6.2: Domain, math\_func, and range for all local grammar rules**

Rule #	Domain	math_func	Range
Rule 1	{Flow}	Number_of_Head_Nodes	{1}
Rule 2	{Material $\cup$ Energy} $\subset$ {Flow}	Number_of_Tail_Nodes	{1}
Rule 3	{Signal} $\subset$ {Flow}	Number_of_Tail_Nodes	{0}
Rule 4	{Function $\cup$ Env} $\subseteq$ Node	Permitted_Input_Types	{ $\neg$ Node}
Rule 5	{Flow}	Permitted_Input_Types	{ $\neg$ Flow}
Rule 6	{Flow}	Number_of_Carrier_Flows	{0, 1}
Rule 7	{Material $\cup$ Energy} $\subset$ {Flow}	Number_of_Carrier_Flows	{0, 1}
Rule 8	{Signal} $\subset$ {Flow}	Number_of_Carrier_Flows	{1}
Rule 9	{Material}	Number_of_Carried_Flows	{1, 2, 3...}
		Subtype_of_Carried_Flows	{Energy $\cup$ Signal }
Rule 10	{Energy}	Number_of_Carried_Flows	{1, 2, 3...}
		Subtype_of_Carried_Flows	{Signal}
Rule 11	{Signal}	Number_of_Carried_Flows	{0}
Rule 12	{{Flow, Flow, Parent-child relation}}	Parent.HeadNode == Child.TailNode	{TRUE}
Rule 13	{Material}	Number_of_Child_Flows	{1, 2, 3...}

<b>Rule #</b>	<b>Domain</b>	<b>math_func</b>	<b>Range</b>
		Type_of_Child_Flows	{Material}
Rule 14	{Material}	Number_of_Parent_Flows	{1, 2, 3...}
		Type_of_Parent_Flows	{Material}
Rule 15	{Material, Material, ...}	Number_of_Parent_Flows	{1, 2, 3...}
		Type_of_Parent_Flows	{Material}
Rule 16	{Energy}	Number_of_Child_Flows	{1, 2, 3...}
		Type_of_Child_Flows	{ Energy }
Rule 17	{Energy}	Number_of_Parent_Flows	{1, 2, 3...}
		Type_of_Parent_Flows	{Energy }
Rule 18	{Energy, Energy, ...}	Number_of_Parent_Flows	{1, 2, 3...}
		Type_of_Parent_Flows	{Energy }
Rule 19	{Signal}	Number_of_Child_Flows	{0}
Rule 20	{Signal}	Number_of_Parent_Flows	{0}
Rule 21	{Material U Energy}	Node_to_Node	{TRUE}
Rule 22	{{Flow, Node}}	Flow.TailNode == Node && Flow.HeadNode == Node	{FALSE}
Rule 23	{Flow}	Env_to_Env	{FALSE}
Rule 24	{Function}	Number_of_Input_Flows	{1, 2, 3...}
		Type_of_Input_Flows	{M U E U S}

Rule #	Domain	math_func	Range
			}
		Number_Of_Output_Flows	{1, 2, 3...}
		Type_of_Output_Flows	{M U E}
Rule 25	{Environment}	Is_Source	{TRUE, FALSE}
		Is_Sink	{TRUE, FALSE}
		Is_Sink && Is_Source	{FALSE}
Rule 26	{{M, E, Carrier relation}}	E.TailNode == M.TailNode	{TRUE}
Rule 27	{{M, E, Carrier relation}}	E.HeadNode == M.HeadNode	{TRUE}
Rule 28	{{S, {M U S}, Carrier relation}}	S.TailNode != NULL	{FALSE}
Rule 29	{{S, {M U S}, Carrier relation}}	S.HeadNode == Carrier.HeadNode	{TRUE, FALSE}
Rule 30	{{Flow, Carrier1, Carrier2}}	Flow.Carrier == Carrier1 && Flow.Carrier == Carrier2	{FALSE}
Rule 31	{{Carried flow, Carrier flow, Carrier relation, Node,}}	Carried.HeadNode == Node && Carrier.HeadNode != Node	{FALSE}

Rule #	Domain	math_func	Range
Rule 32	{{Carried flow, Carrier flow, Carrier relation, Node,}}	Carried.TailNode == Node && Carrier.TailNode != Node	{FALSE}
Rule 33	{Flow}	Flow.Carrier == Flow	{FALSE}

It can be seen that rule pairs such as Rule 6 and Rule 7 meet the necessary condition for conflict, as they have the same `math_func` and intersecting domains. However, in those cases, the range is also identical, which violates the sufficient condition for conflict. Similarly, Rule 6 and Rule 8 have the same necessary condition, as `{Signal}` is a subset of `{Flow}`. However, the range of Rule 8, `{1}`, is also a subset of the range of Rule 6, `{0, 1}`, thus avoiding conflict.

Based on these examinations, it is seen that there is no pair of rules that satisfy the necessary and sufficient conditions for conflict. For most cases, the necessary condition is not met, as the rules do not have the same `math_func` and intersecting domains. In the few cases where this necessary condition is met, the ranges are also consistent, thus violating the sufficient condition of conflict. **Based on this analysis, it is concluded that the set of rules in the local grammar are consistent.** This conclusion is also indirectly supported by the lack of repetition of modeling constructs in the tables in Chapter 5. There are a total of 86 constructs in Chapter 5. However, no two of them are topologically identical. Since each rule solves a different problem, the modeling constructs where their effect is pertinent are also different. Notably, the constructs were

not carefully made to look unique. Their uniqueness is a result of the consistency of the rules.

Ultimately, Sections 6.1.1 and 6.1.2 collectively support three claims: (1) all constructs needed for modeling a function structure are captured in the tables in Section 5.2, (2) all rules needed to determine if each of those constructs is permitted or prohibited are available in Section 5.2, and (3) the rules thus available are mutually consistent. **Thus, it is shown that the local grammar of the first layer of the representation is internally consistent.**

### **6.1.3 Ontological Examination of Consistency of the Vocabulary**

To test consistency of the vocabulary, the class definitions from Chapter 5 are committed to an ontology using the Protégé Frames Ontology Editor<sup>12</sup> and then checked using the Protégé consistency checker. An ontology is an explicit specification of domain of discourse [27-29, 66, 172, 173]. Information is organized in terms of the entities comprising the domain, the properties used to characterize the concepts, relations between those concepts, and constraints imposed on those concepts, relations, and properties [172]. In this sense, an ontology holds the same information types as a formal representation, defined in Section 1.2.2. It is thus reasonable to expect that the Conservation Layer (Layer 1) of the formal representation presented in Chapter 5 should

---

<sup>12</sup> Available at <http://protege.stanford.edu/overview/protege-owl.html>, accessed on August 16, 2011

be implementable in ontological form that satisfies the consistency checks. The Web Ontology Language (OWL) is a family of languages for authoring ontologies and explicitly capturing formal semantics of concepts, especially for semantic reasoning [174]. Protégé is a free, open-source software that supports ontology-editing with OWL and conforms to requirements of data exchange (e.g., XML), notations (e.g., OWL), and frameworks (e.g., RDF) necessary for developing semantic networks, as set by the World Wide Web Consortium, W3C<sup>13</sup>. However, the main purpose of using Protégé OWL in this research is to use its reasoning ability to check consistency of the asserted ontology and to draw inferences about the ontological identity of classes and objects. For example, the knowledge that a function can produce only material and energy flows, but not signals, can be asserted by setting the domain of the relation TailNode as {Material U Energy}. Thereafter, if the asserted description of the class Signal or any of its instances includes a TailNode property, the Protégé reasoner detects these two assertions to be in conflict. Further, the Protégé reasoner can determine if a class or an instance asserted by the user can be interpreted as another class or its instance, based on the properties declared in the instance – an ability that can be used to check if the asserted concepts (classes, properties, restrictions) are ambiguous. The construction of the ontology is presented next with figures and text. A computer code for reconstructing this ontology in

---

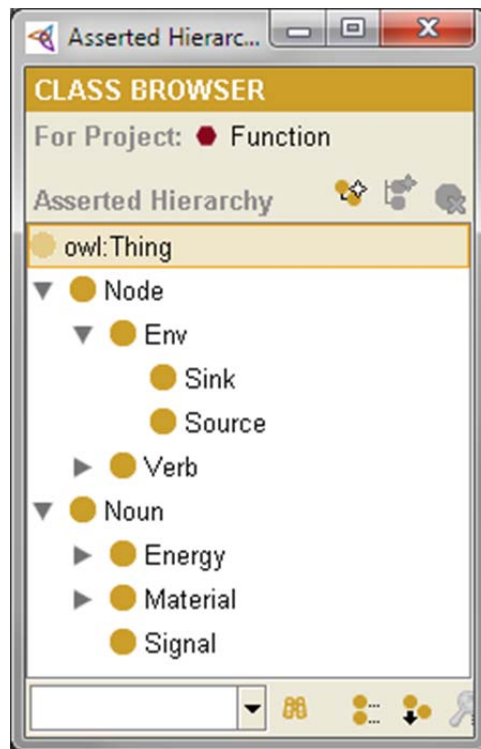
<sup>13</sup> <http://www.w3.org/>, accessed on August 16, 2011



Protégé OWL is available in Appendix B in the Extensible Markup Language (XML) format.

### Asserted Class Hierarchy and Sibling Disjunction

Figure 6.1 shows the asserted class hierarchy in the ontology.

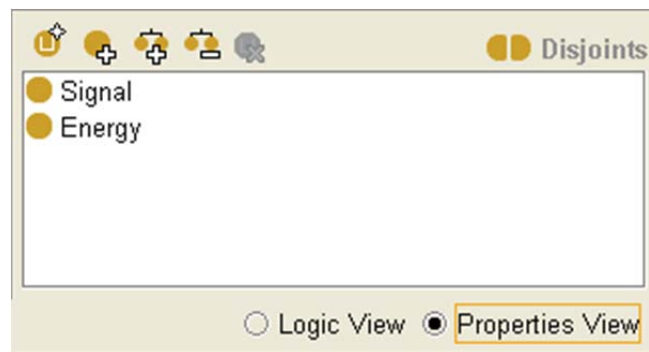


**Figure 6.1: OWL class hierarchy (Asserted)**

The hierarchy is identical to the vocabulary presented in Table 5.1, except that the OWL class owl:Thing is used instead of the top-level entity Element. Owl:Thing is the mandatory top-level class enforced by Protégé. Adding Element under owl:Thing and adding the remaining hierarchy under Element would produce an equally consistent ontology, although it is unnecessary to do so. For this ontology's purpose, owl:Thing is

equivalent to Element. Further, Verb, Energy, and Material have more hierarchical levels under them (collapsed). These levels are used in the subsequent layers of the representation and are not exposed here since the scope of this chapter is limited to validating the first layer only.

During creating the class hierarchy, all subclasses under a superclass (called sibling classes in OWL terminology) are asserted to be disjoint, thus capturing the exclusive disjunction (XOR) relations in the ERA diagram (Figure 5.1). Thus, Node and Noun are mutually disjoint, Environment and Verb are mutually disjoint, Material, Energy, and Signal are mutually disjoint, and Source and Sink are mutually disjoint. For illustration, the disjoint assertions for the class Material are shown in Figure 6.2.



**Figure 6.2: Exclusive disjunction between Material, Energy, and Signal (Asserted)**

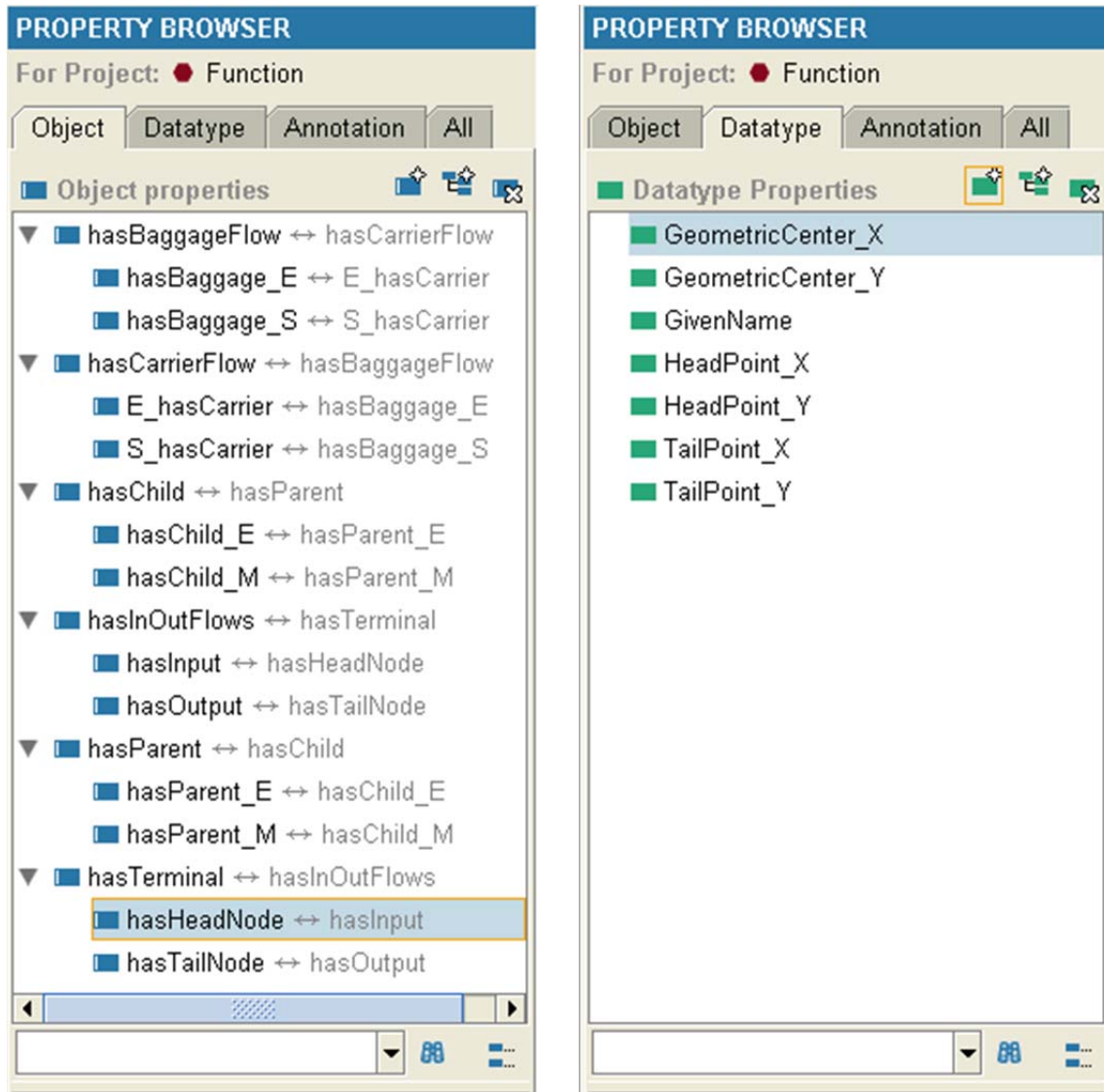
#### Asserted Object Properties and Data Properties

Object properties in Protégé are analogous to relations between the classes. Data properties are analogous to the attributes. Figure 6.3 shows the asserted object properties and data properties in the ontology. The properties replicate the relations shown in the ERA diagram in Figure 5.1. The `hasHeadNode` and `hasTailNode` properties capture the

HeadNode and TailNode relations, hasChild captures the Child relations, and hasCarrier represents the CarrierFlow relations. In addition, by exploiting the benefits of Protégé OWL, the inverse properties of each property are captured, as indicated by the  $\leftrightarrow$  symbols. For example, the fact that a flow's head is connected to a node can be described in two ways: (1) by asserting that the flow's head node (hasHeadNode property) is the node or (2) by asserting that one of the node's input flows (hasInput property) is the flow. These two assertions are equivalent, they capture the same information about the model topology, yet the first is written as a flow property and the second is a node property. These two properties are the inverse property of each other.

Except for the hasCarrierFlow and hasBaggageFlow property groups, the object properties are organized in a hierarchical manner in order to facilitate their conceptual grouping. The super-properties are not directly used in model creation, as no domain or range is necessary or asserted for them. For the carrier and baggage properties, the child properties are created because Protégé does not allow defining multiple sets of domains and ranges for the same property name. For example, although the restriction of flow subtypes for carrier relations can be controlled by unary grammar rules (Table 5.14), separate relations must be created for capturing those restrictions. In this case, the E\_hasCarrier property can connect any Energy (domain) to any Material (range), while the S\_hasCarrier property has a domain of Signal flows and a range of Material U Energy, thus capturing the unary carrier-carried rules of Table 5.14. The domain and range of the properties are discussed in the next section. The data properties capture the attributes mentioned in the ERA model. Since Protégé OWL allows only basic data types

such as integer, floating points, and strings, the point coordinates for geometric center, head point, and tail point are expressed as individual X and Y coordinates (floating point numbers).



(a) Object properties

(b) Data Properties

Figure 6.3: Object properties and data properties (Asserted)

It should be noted that the apparent disagreement between the attributes and relations of the ERA model and the properties in this ontology, such as the necessity to use the hierarchical properties for the carrier relation and the floating point numbers for the point coordinates, are not indicative of inconsistency of the ERA model. Rather these are results of the implementation environment provided by Protégé OWL. In the next section, the representation is implemented in an object-oriented application for external validation. In that implementation, the points are captured as instances of a CPoint class and a single property is used to capture the carrier-carried relations between different flow subtypes, using algorithmic enforcement of the unary carrier-carried rules of Table 5.14.

#### Asserted Domain, Range, and other Qualifiers

Domain and range of a property in Protégé are analogous to the entities connected by a property in the ERA model. Figure 6.4 shows a Protégé screenshot of the property list, along with their domains and ranges. The domain of a property is the set of things that can “have” that property or for which that property is defined. For example, since only Source and Verb instances are the only things that can produce output flows, the domain of property hasOutput (highlighted row in the figure) is Source U Verb. The range of a property is the set of things that the property can point to, or the set that can be related to the domain through the property. For example, since only Material and Energy flow instances can be related to a Verb or Source instance through the hasOutput property, the range of hasOutput is Material U Energy. In this manner, the asserted domains and ranges reflect the relations in the ERA model and also some grammar rules.

Name	Range	Domain	Other Characteristics
E_hasCarrier	Material	Energy	[...] Functional, Transitive, Super properties: {hasCarrierFlow}
GeometricCenter_X	float	owl:Thing	Functional
GeometricCenter_Y	float	owl:Thing	Functional
GivenName	string	owl:Thing	Functional
HeadPoint_X	float	Noun	Functional
HeadPoint_Y	float	Noun	Functional
S_hasCarrier	Material or Energy	Signal	[...] Functional, Transitive, Super properties: {hasCarrierFlow}
TailPoint_X	float	Noun	Functional
TailPoint_Y	float	Noun	Functional
hasBaggageFlow		owl:Thing	[...]
hasBaggage_E	Energy	Material	[...] InverseFunctional, Super properties: {hasBaggageFlow}
hasBaggage_S	Signal	Material $\sqcup$ Energy	[...] InverseFunctional, Super properties: {hasBaggageFlow}
hasCarrierFlow		owl:Thing	[...]
hasChild		owl:Thing	[...] Transitive
hasChild_E	Energy	Energy	[...] Transitive, Super properties: {hasChild}
hasChild_M	Material	Material	[...] Transitive, Super properties: {hasChild}
hasHeadNode	Verb or Sink	Noun	[...] Functional, Super properties: {hasTerminal}
hasInOutFlows		owl:Thing	[...]
hasInput	Noun	Verb $\sqcup$ Sink	[...] InverseFunctional, Super properties: {hasInOutFlows}
hasOutput	Material or Energy	Source $\sqcup$ Verb	[...] InverseFunctional, Super properties: {hasInOutFlows}
hasParent		owl:Thing	[...] Transitive
hasParent_E	Energy	Energy	[...] Transitive, Super properties: {hasParent}
hasParent_M	Material	Material	[...] Transitive, Super properties: {hasParent}
hasTailNode	Source or Verb	Material $\sqcup$ Energy	[...] Functional, Super properties: {hasTerminal}
hasTerminal		owl:Thing	[...]

**Figure 6.4: Domain, range, and other characteristics of properties (Asserted)**

In addition to the domain and range information, three special characters are asserted for each property. A property is called functional if for it points to one specific instance within the range. For example, since a flow can have only one tail node, the `hasTailNode` property is marked as Functional. A property is called inverse functional if its inverse property is functional. For example, since `hasTailNode` is functional, its inverse property `hasOutput` is inverse functional. A property is transitive if the two assertions (1) instance A is related to B and (2) B is related to C imply that A is related to C. For example, the `hasChild` properties are transitive, since they imply derivation through conservation, in which case if A is derived from B and B is derived from C, it is correct to infer that A is derived (indirectly) from C.

### Asserted Restrictions (Axioms)

Restrictions in Protégé are analogous to the local grammar rules that determine which properties (relations) can exist between the classes. Three types of restrictions—existential, universal, and cardinal—are possible in Protégé OWL. For illustration, Figure 6.5 shows the restrictions asserted on the properties of the class Energy. An existential restriction is used to assert that an instance must have a property. For example, the assertion of domain and range for the hasTailNode property ensures that an Energy instance (in domain) “can” have a tail node, which must be a Source or a Verb instance. The existential restriction on this property asserts that an Energy instance “must” have “at least one” tail node within that range. However, since the hasTailNode property is also asserted as functional, there can be one and only one tail node for an Energy instance. The cardinality restriction is used to set exact, upper, or lower bound of a property. For example, the domain and range of the E\_hasCarrier property assert that there “can” be one and only one (functional) carrier flow of an Energy flow (domain), which must be a Material (range). The cardinality of “ $\leq 1$ ” in this case asserts that an Energy flow can have at most one carrier. In this manner, restrictions are asserted for each property of each class, as applicable. The Protégé OWL classes inherit restrictions from their superclasses. For example, the exact cardinality of the X and Y coordinates of the head and tail points of Energy are inherited from its superclass Noun, where they are asserted one time so that they can be applied to all Noun subclasses.



**Figure 6.5: Restrictions on Energy (Asserted and Inferred)**

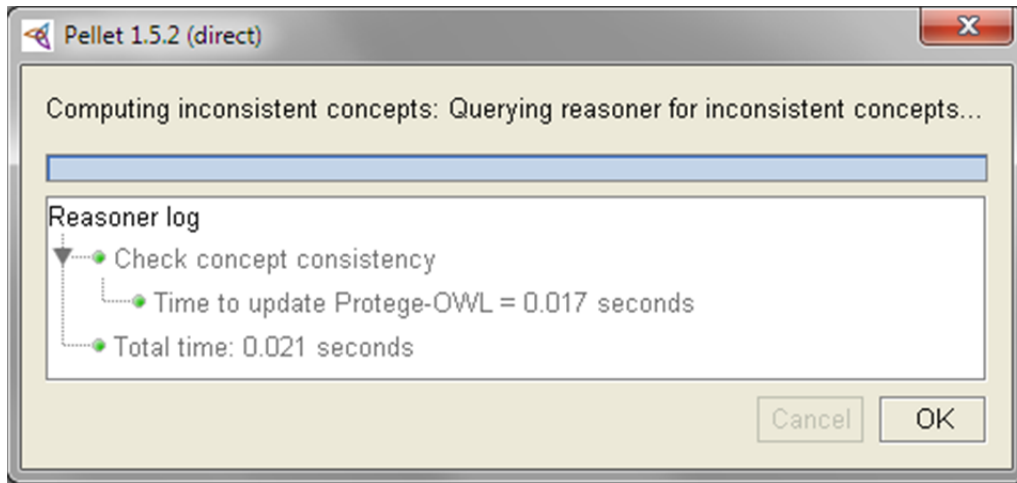
With the assertion of the classes and their hierarchy, the object and data properties, their domains, ranges, and qualifiers, and the restrictions, the assertion of the ontology is complete. As a whole, the ontology replicates the entities, relations, attributes, and the applicable grammar rules that can be replicated using the ontological environment. At this point, the reasoner is used to check consistency of the asserted ontology.

#### Ontological Reasoning for Consistency Checking and Ambiguity

The Pellet 1.5.2 reasoner [175] is used to examine the ontology. Figure 6.6 shows the outcome of the consistency checker. The results prove that the asserted ontology is consistent, as inconsistent ontological elements, if any, are reported in red color in this

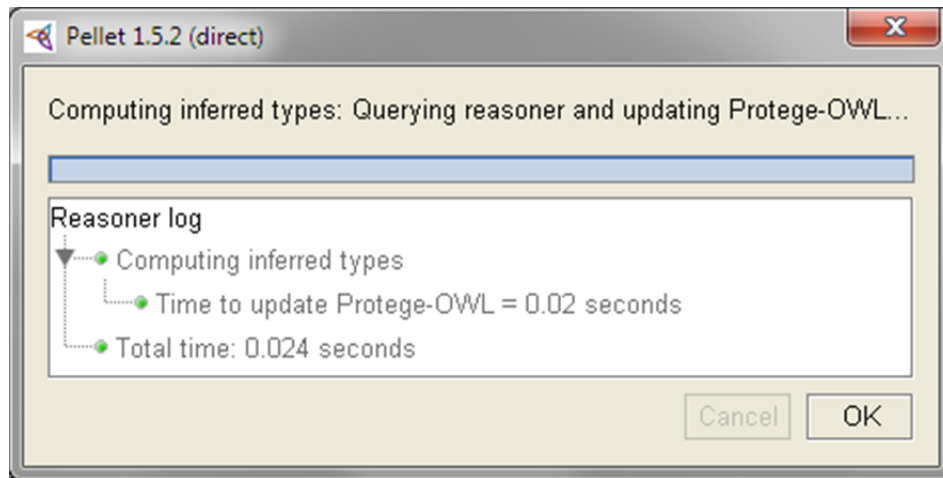


output window. This illustration verifies that no conflict exists between the asserted statements.

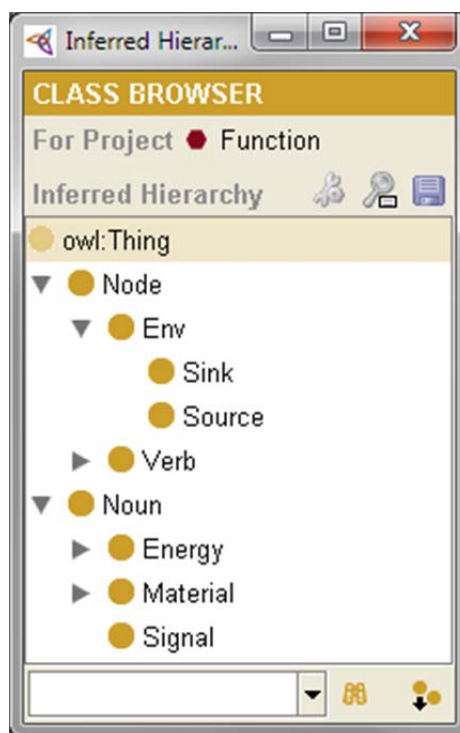


**Figure 6.6: Consistency checking results for the ontology (Consistent)**

Figure 6.7 shows the outcome of the reasoner for computing inferred data types and Figure 6.8 shows the class hierarchy inferred by the reasoner from the asserted hierarchy. No data type except the explicitly asserted ones is found. The inferred and asserted class hierarchies are also identical. This indicates that each concept in the ontology is unambiguously defined, as there is no possibility of inferring one concept as another.



**Figure 6.7: Consistency checking results for the ontology (Consistent)**



**Figure 6.8: OWL class hierarchy (Inferred) – Identical with the asserted hierarchy**

## Model Construction and Model-Level Consistency Checking using the Ontology

Based on the above three illustrations of consistency of the classes, it is expected that instances created from those classes and related by the asserted properties will define a model that will also be consistent. To this end, the function structure generated at the final step of the modeling exercise (Model State 4.14) is created using the ontology. Figure 6.9 shows a Protégé screenshot of the Verb instance En\_Air\_3, for illustration. This instance represents the function block Energize Air 3 in Model State 4.14. The total number of instances created for each class is shown beside each class name in the left panel (Class Browser) of this figure. These numbers can reflect the number of instances in Model State 4.14. For verbs, the eight instances are seen in the center panel (Instance Browser). The form for adding a verb asks exactly those questions that are necessary to define a verb: its input flows (hasInput) and output flows (hasOutput). These entries replicate the topology of Model State 4.14.

Upon finishing the model using the instances of nodes, flows and their topological relations, the consistency checker is invoked again, to test the model-level consistency and to ensure that while the representation itself is consistent, it can support consistent models. Similar to Figure 6.6, the reasoner does not detect any inconsistency in the model. **Based on consistency of both the ontology and the model, the internal consistency of the vocabulary is now demonstrated. Further, based on the demonstration of exhaustiveness and consistency of the local grammar by logical examination and consistency of the classes through this ontological examination, it is**

asserted that the internal consistency of the representation as a whole is now demonstrated.

While the representation is consistent, it remains to be validated that it can actually support the reasoning it is designed for: (1) topologic and (2) derivational reasoning at a qualitative level. This external validation is addressed next.

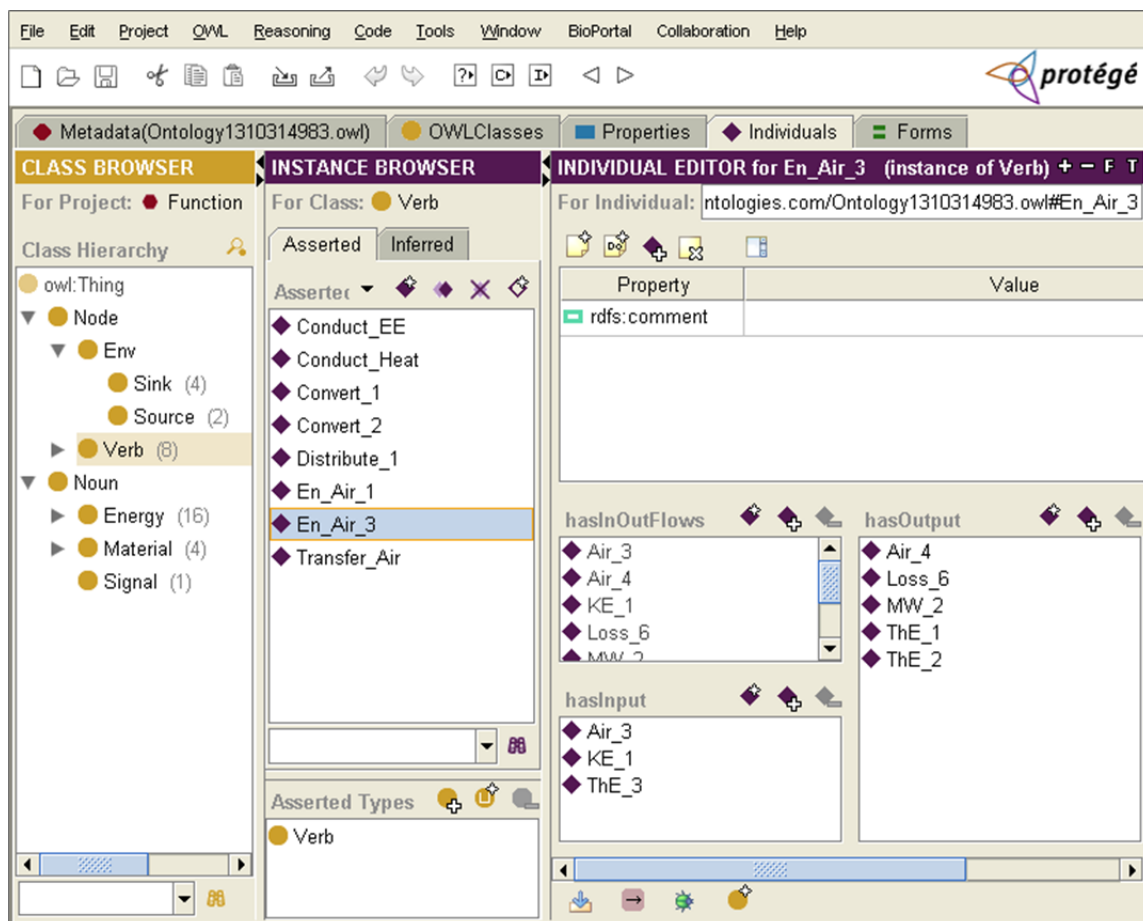


Figure 6.9: Creation of individual instances using the ontology

## **6.2 Demonstration of External Validity against Conservation Laws**

External validity of the representation is demonstrated by incorporating the representation in an object-oriented, graphic user interface-based software tool, using the tool to construct models, and reason on them using the laws of conservation. The design of this software tool is discussed next. The software tool is named the Concept Modeler, ConMod for short, as the representation is intended to support modeling and reasoning on early design concepts.

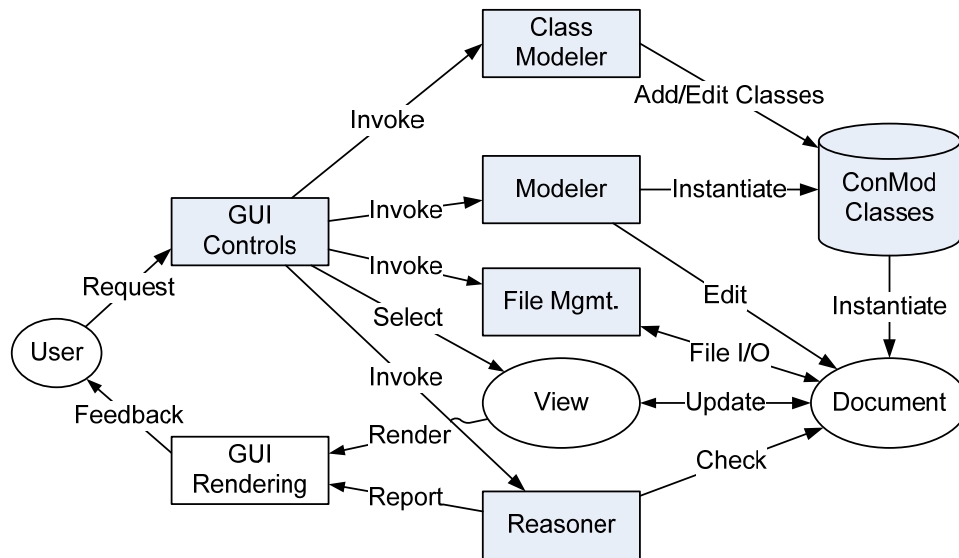
### **6.2.1 Design of the Software Tool ConMod**

The ConMod program is implemented in order to demonstrate external validity of the proposed representation against the laws of physics—specifically, conservation laws for the first layer—by showing that the reasoning needs identified in Chapter 4 are indeed supported by the representation. The tool itself is not a direct outcome of the research. It is only a means to validate that the direct outcome of this research—the representation—is valid. The tool is developed using the Microsoft Foundation Classes (MFC) library that provides Application-Programmer Interface (API) classes in the C++ language for writing Windows-based programs. The default version of an MFC project, when compiled, creates a Windows application with empty windows, basic toolbar buttons, and menu items such as File Open, File Close, Save, Cut, Copy, Paste, Print, and Help, while each of these buttons and menus are nonfunctional. Code is written to implement functionality for these buttons and menus, such as model construction, editing, and reasoning. The high-level system architecture for ConMod is discussed next.

It must be emphasized that the sole purpose of this software implementation is to allow external validation through automated reasoning. In order to perform reasoning, a computer model is first required. The software is designed only to provide models for this illustration and to execute the reasoning algorithms. The design of the software, such as its user interface, is not examined here. Only the minimal features required to demonstrate reasoning are implemented.

### ConMod System Architecture

The system architecture of ConMod is shown in Figure 6.10. This architecture is based on the Document-View architecture provided by MFC. Two overridable MFC classes—document and view—are used to hold data. The document class (CConModDoc in source code) is instantiated only once per session and holds the instances of model classes such as functions and flows. The view class (CConModView) holds the graphic data to render the model on screen and uses private methods to automatically update whenever there is a change in the document due to a model edit.



**Figure 6.10: The document-view architecture of ConMod**

There are six individual functional modules of the application, indicated by shaded shapes in the figure.

1. **GUI Controls:** This module implements the toolbars, buttons, menu items, and context menu lists and serves as the user's gateway to the application. It invokes the other modules and helps the user select instances from the view.
2. **ConMod Classes:** This module is the static repository where the classes are declared and implemented. Data in this module is used in constructing the instances, but is never changed during modeling or reasoning operations in a session of the application. Data in this module can be edited only by the Class Modeler module. Classes in this repository also contain the methods for construction, rendering, and edit operations pertinent to the classes, which are called by the Modeler during model edits.

3. **Modeler:** This module includes the methods for adding, editing, and deleting instances of ConMod classes such as Verb, Noun, and Environment. Instances are created and edited on the document, which causes the view to automatically update the rendering on screen.
4. **File Management:** This module is used to perform file input output operations, such as saving and launching. During saving, data necessary to reconstruct the model are read from the document and saved in a file (not shown). The extension for ConMod model files is \*.fst, which stands for “function structure”. During launching (File Open), data from the file is read and added to the document, which causes the view to update.
5. **Reasoner:** This module is invoked by the GUI at user request. It contains and executes the algorithms required to check the model for qualitative conservation. It only reads the instances in the document and reports the findings to the GUI rendering module, to be returned to the user.
6. **Class Modeler:** This module is available for defining new function and flow classes by the designer or modeler without requiring editing the source code, thus allowing capture and reuse of domain-specific design knowledge through customization.

The functionality of these modules is implemented in the source code, available in Appendix C and Appendix D. The executable file for the program, ConMod.exe, is available upon request from the Clemson Engineering Design Applications and Research



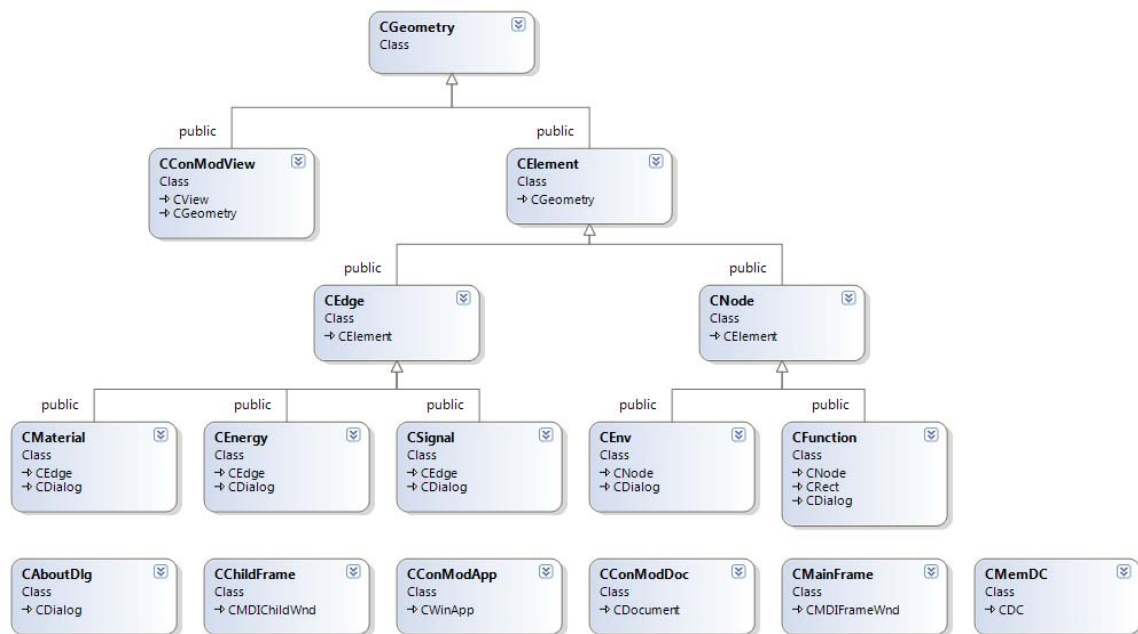
Lab, CEDAR<sup>14</sup>. The class diagram and description for this application are discussed next.

### ConMod Class Diagram and Class Descriptions

Figure 6.11 shows the class diagram for ConMod. Classes from the vocabulary of Table 5.1 are seen in the subclasses under CElement. The class Noun is named CEdge and Verb is named CFunction in this implementation. The shown version of ConMod is a partial implementation of the representation, where classes Source and Sink are not shown. This simplification does not compromise the demonstration of the conservation reasoning, as illustrated in the next sections.

---

<sup>14</sup> <http://www.clemson.edu/ces/cedar>, accessed on August 16, 2011



**Figure 6.11: Class diagram of the ConMod application**

The classes at the bottom are MFC classes that provide general functionality for Windows-based applications such as windows, frames, dialogs, and device contexts (canvas for drawing objects). The only exception is CConModDoc, the document class that holds the instances added to a model. The CGeometry class is required at the highest level as many classes underneath it use geometric data and methods provided by it for drawing instances.

The complete source code of this application is available in Appendix C and Appendix D. For inspection of the implementation of the data members and methods in the code, the declarations within the classes (C++ header files) are presented in Table 6.3 through Table 6.13. The purpose is to illustrate that the code actually implements the class data elements and methods identified in the representation and to explain occasional

deviations where that adherence is violated. Data members and methods that pertain to the software design aspect such as the user interface and dialogs, and those that are defined in the subsequent layers such as irreversibility, are omitted for brevity. These data and methods do not interfere with the conservation reasoning illustrated.

In each table, the class declaration in the header file is shown to illustrate its superclasses. The data members and member functions are then described. For data members, the first word is the data type and the second word is the name of the data member. For methods, the first word before the parentheses is the name of the method, the words to the left of the name are the return data type and the virtual qualifier (optional), and the list in the parentheses contains the data passed to the method as arguments. Each argument is described with its data type (first word), name (second word), and optionally with its default value (assignment operation with “=” sign). Table 6.3 describes the highest-level class CGeometry.

**Table 6.3: Class CGeometry**

<b>Class declaration</b>	
<pre>class CGeometry // Not inherited from anything - top level class</pre>	
<b>Data members</b>	<b>Description</b>
None	
<b>Methods (Member functions)</b>	<b>Description</b>
<pre>int RoundToTwenty(float n, int t);</pre>	Rounds up n to the nearest multiple of t

	and returns the result as an integer
<code>CPoint SnapToGrid(CPoint p);</code>	Computes a point derived by snapping the input point p to the grid size, a global variable
<code>long distance(CPoint p1, CPoint p2);</code>	Computes the distance between two points, p1 and p2
<code>CPoint* InterpolatePoints(CPoint p1, CPoint p2, double ratio);</code>	Computes a point by interpolating between two input points, p1 and p2, by the input ratio

Table 6.4 describes the class CElement, analogous to the class Element in the design (Table 5.2).

**Table 6.4: Class CElement (Element)**

<b>Class declaration</b>	
<code>class CElement : public CGeometry</code> <code>// Inherits CGeometry, otherwise the top-level class for elements</code>	
<b>Data members</b>	<b>Description</b>
<code>// Geometric and name attribute</code> <code>data</code>	
<code>CString GivenName;</code>	Name given to an element by the user
<code>CPoint GeometricCenter;</code>	Holds the geometric center of an element
<code>CPoint AnchorsForBaggageFlows[16];</code>	A list of points where carrier flow ends

	can be connected to the stem of a flow arrow
<code>// Input-output relation data</code>	
<code>CPoint TailPoint, HeadPoint;</code>	Point data members storing the location of the head and tail of point flows
<code>CElement* pHeadElem;</code>	Pointer to the element that is the head element of a flow instance
<code>CElement* pTailElem;</code>	Pointer to the element that is the tail element of a flow instance
<code>CPoint Anchors[16];</code>	A list of points where flows can be connected to a function or environment, or nodes can be connected to a flow (head and tail points)
<code>// Model editing data</code>	
<code>bool IsHighlighted;</code>	Flag set to true when an element is highlighted for selection by the user
<code>bool IsSelected;</code>	Flag set to true when an element is selected for an edit operation by the user
<code>int GrabHandle;</code>	An integer that stores where an element is “grabbed” by the mouse: tail, center, or head
<code>// Drawing data - pen and brush</code>	

<code>int PenR, PenG, PenB;</code>	Color settings for the drawing pen
<code>int BrushR, BrushG, BrushB;</code>	Color setting for the drawing brush
<code>int HeadBrushR, HeadBrushG, HeadBrushB;</code>	Color setting for the brush that paints the head of a flow by topological status (dangling or attached)
<code>int TailBrushR, TailBrushG, TailBrushB;</code>	Color setting for the brush that paints the tail of a flow by topological status (dangling or attached)
<b>Methods (Member functions)</b>	<b>Description</b>
<code>virtual void DrawOnDC (CDC* pDC) ;</code>	Method for drawing individual instances on screen, declared as virtual as it is differently implemented for different element types

Table 6.5 describes the class CNode, analogous to the class Node in Table 5.2.

**Table 6.5: Class CNode (Node)**

<b>Class declaration</b>	
<code>class CNode : public CElement</code> <code>// Inherits CElement per the design of the vocabulary</code>	
<b>Data members</b>	<b>Description</b>
<code>// Input-output relations</code>	
<code>bool NoInputAttached;</code>	Flag to detect if the node has no input

	flows
<code>bool NoOutputAttached;</code>	Flag to detect if the node has no output flows
<b>Methods (Member functions)</b>	<b>Description</b>
<code>void ComputeBlockCoordinates();</code>	Method to compute the node vertices – overridden by CFunction and CEnv

Table 6.6 describes the class CEdge, analogous to the class Noun in Table 5.2.

**Table 6.6: Class CEdge (Noun)**

<b>Class declaration</b>	
<pre>class CEdge : public CElement // Inherits CElement (equivalent to Noun) per the design of the vocabulary</pre>	
<b>Data members</b>	<b>Description</b>
<code>// Input-output relations</code>	
<code>void ComputeAnchorPoints();</code>	Method to compute the anchor points for a flow, including the head and tail points and the points on the stem for attaching carried flows
<code>void AttachEdgeToNearestAnchor();</code>	Method to attach an edge to the nearest anchor available in the whole model, helping the user to connect an edge end to

	an element
<code>void ResetGeometricCenter();</code>	Method to recomputed the geometric center of an edge after it is moved or reconnected
<code>// Carrier-carried relations</code>	
<code>bool ThisFlowIsIncomingBaggage;</code>	Boolean to indicate if the flow is added as a baggage (carried) to another flow
<code>bool ThisFlowIsOutgoingBaggage;</code>	Boolean to indicate if the flow is carried by a another flow and used by a function
<code>// Head drawing data</code>	
<code>CPoint HeadLeftVertex, HeadRightVertex;</code>	Points to hold the left and right vertices of the head of a flow
<code>double HeadSize, HalfHeadAngle;</code>	Numbers to contain the head size in pixels and the half angle of the arrow head
<code>CPoint HeadVertexArray[3];</code>	Array holding the three points defining the head of a flow as a triangle
<code>// Stem drawing data</code>	
<code>int StemThickness;</code>	Integer showing the thickness of a flow that carries between Material, Energy, and Signal
<code>int StemLineFont;</code>	Font style of the flow stem that varies between Material, Energy, and Signal



<code>int FontSize;</code>	Font size of the flow name on the stem, which varies between carrier and carried flows
<b>Methods (Member functions)</b>	<b>Description</b>
<code>// Flow constructor CEdge(CPoint TailClick, CPoint HeadClick);</code>	Constructor method that adds a flow using the two points on the screen where the mouse is clicked as input for the tail and head
<code>// Drawing method void DrawOnDC(CDC* pDC);</code>	Method to draw a flow with its stem, its tail circle, its head with three lines, and fill colors in the head and tail ac per their dangling status

Table 6.7 describes the class CFunction, analogous to the class Verb in Table 5.2.

**Table 6.7: Class CFunction (Verb)**

<b>Class declaration</b>	
<code>class CFunction : public CNode, public CRect, public CDialog  // Inherits CNode per the design of the vocabulary  // Inherits the MFC class CDialog as a dilaog isrequired to accept user input  // Inherits the MFC class CRect to facilitate drawing function blocks</code>	
<b>Data members</b>	<b>Description</b>
<code>// Attributes</code>	

<code>CString GivenName;</code>	Name given to the function by the modeler
<b>Methods (Member functions)</b>	<b>Description</b>
<code>void ComputeBlockCoordinates();</code>	Method to compute the vertices of the function block
<code>void DrawOnDC(CDC* pDC);</code>	Method to draw the function blocks

Table 6.8 describes the class CEnv, analogous to Environment in Table 5.2.

**Table 6.8: Class CEnv (Environment)**

<b>Class declaration</b>	
<pre>class CEnv : public CNode, public CDialog // Inherits CNode per the design of the vocabulary // Inherits the MFC class CDialog as a dialog is required to accept user input</pre>	
<b>Data members</b>	<b>Description</b>
<code>// Attributes</code>	
<code>CString GivenName;</code>	Name given to the environment instance by the modeler
<b>Methods (Member functions)</b>	<b>Description</b>
<code>void ComputeBlockCoordinates();</code>	Method to compute the vertices of the environment hexagon
<code>void DrawOnDC(CDC* pDC);</code>	Method to draw the environment

	hexagons
--	----------

Table 6.9 describes the class CMaterial, analogous to class Material in Table 5.2.

**Table 6.9: Class CMaterial (Material)**

<b>Class declaration</b>	
<pre>class CMaterial : public CEdge, public CDialog // Inherits CEdge per the design of the vocabulary // Inherits the MFC class CDialog as a dialog is required to accept user input</pre>	
<b>Data members</b>	<b>Description</b>
<i>// Attributes</i>	
CString GivenName;	Name given to the environment instance by the modeler
<i>// Parent-child relations</i>	
CList<CMaterial*, CMaterial*> ChildList;	List of pointers to child flows, all of which are instances of class CMaterial
CList<CMaterial*, CMaterial*> ParentList;	List of pointers to parent flows, all of which are instances of class CMaterial
<b>Methods (Member functions)</b>	<b>Description</b>
void DrawOnDC(CDC* pDC);	Method to draw the CMaterial instance

Table 6.10 describes the class CEnergy, analogous to the class Energy in Table 5.2.

**Table 6.10: Class CEnergy (Energy)**

<b>Class declaration</b>	
<pre>class CEnergy : public CEdge, public CDialog // Inherits CEdge per the design of the vocabulary // Inherits the MFC class CDialog as a dilaog isrequired to accept user input</pre>	
<b>Data members</b>	<b>Description</b>
<pre>// Attributes</pre>	
<pre>CString GivenName;</pre>	Name given to the environment instance by the modeler
<pre>// Parent-child relations</pre>	
<pre>CList&lt; CEnergy*, CEnergy*&gt; ChildList;</pre>	List of pointers to child flows, all of which are instances of class CEnergy
<pre>CList&lt; CEnergy*, CEnergy*&gt; ParentList;</pre>	List of pointers to parent flows, all of which are instances of class CEnergy
<b>Methods (Member functions)</b>	<b>Description</b>
<pre>void DrawOnDC(CDC* pDC);</pre>	Method to draw the CEnergy instance

Table 6.11 describes the class CSignal, analogous to the class Signal in Table 5.2.

**Table 6.11: Class CSignal (Signal)**

<b>Class declaration</b>	
<pre>class CSignal : public CEdge, public CDialog // Inherits CEdge per the design of the vocabulary</pre>	

<code>// Inherits the MFC class CDialog as a dilaog isrequired to accept user input</code>	
<b>Data members</b>	<b>Description</b>
<code>// Attributes</code>	
<code>CString GivenName;</code>	Name given to the environment instance by the modeler
<code>// Parent-child relations - NONE</code>	
<b>Methods (Member functions)</b>	<b>Description</b>
<code>void DrawOnDC (CDC* pDC);</code>	Method to draw the CEnergy instance

Table 6.12 describes the MFC document class CConModDoc. This class is not included in the vocabulary of Table 5.2. It is used to hold the model instances in this implementation.

**Table 6.12: Class CConModDoc (Document)**

<b>Class declaration</b>	
<code>class CConModDoc : public CDocument</code>  <code>// This class is not a part of the vocabulary of the representation and thus does not inherit anything from that vocabulary</code>  <code>// Inherits the MFC class CDcoument to store model elements and update views</code>	
<b>Data members</b>	<b>Description</b>
<code>CList&lt;CElement*, CElement*&gt;</code> <code>CElementList;</code>	List of all CElement objects, irrespective of types. Used by methods that do not depend

	on the types, such as deleting and refreshing the screen
<code>CList&lt;CNode*, CNode*&gt; CNodeList;</code>	List of all nodes
<code>CList&lt;CEdge*, CEdge*&gt; CEdgeList;</code>	List of all flows
<code>CList&lt;CFunction*, CFunction*&gt; CFunctionList;</code>	List of only functions
<code>CList&lt;CEnv*, CEnv*&gt; CEnvList;</code>	List of only environment instances
<code>CList&lt;CMaterial*, CMaterial*&gt; CMaterialList;</code>	List of only material flows
<code>CList&lt;CEnergy*, CEnergy*&gt; CEnergyList;</code>	List of only energy flows
<code>CList&lt;CSignal*, CSignal*&gt; CSignalList;</code>	List of only signal flows
<code>CList&lt;CElement*, CElement*&gt; PreselectionList;</code>	List of elements (irrespective of types) that are highlighted for an operation
<b>Methods (Member functions)</b>	<b>Description</b>
None	

Table 6.13 shows the MFC view class, `CConModView`, used to create the graphics of the model and to support the reasoning algorithms.

**Table 6.13: Class `CConModView` (View)**

<b>Class declaration</b>
<pre>class CConModView : public CView, public CGeometry // This class is not a part of the vocabulary of the representation and</pre>

<pre>thus does not inherit anything from that vocabulary  // Inherits the MFC class CDcoument to store model elements and update views  // Inherits geometry as it needs geometric functions to update the view</pre>	
<b>Data members</b>	<b>Description</b>
<pre>// Model reasoning data members</pre>	
<pre>CString Msg_OrphanFlow;</pre>	String that holds the orphan flow message
<pre>CString Msg_BarrenFlow</pre>	String that holds the barren flow message
<pre>CString Msg_OneInManyOut_M;</pre>	String that holds the error message when material conservation is violated in the one-in-many-out configuration
<pre>CString Msg_OneInManyOut_E;</pre>	String that holds the error message when energy conservation is violated in the one-in-many-out configuration
<pre>CString Msg_ManyInOneOut_M;</pre>	String that holds the error message when material conservation is violated in the many-in-one-out configuration
<pre>CString Msg_ManyInOneOut_E;</pre>	String that holds the error message when energy conservation is violated in the many-in-one-out configuration
<pre>CString Msg_ManyInManyOut;</pre>	String that holds the error message telling that unless derivation relations are shown,

	it is not possible to conclude on a violation in the many-in-many-out configuration
<b>Methods (Member functions)</b>	<b>Description</b>
<code>// Screen refresh</code>	
<code>virtual void OnDraw(CDC* pDC);</code>	Method to redraw every element instance in the model to refresh the whole screen
<code>// Model construction methods</code>	
<code>void AddFunction(void);</code>	Method to add a function to the model
<code>void AddEnv(void);</code>	Method to add an environment to the model
<code>void AddEdge_Dynamic(void);</code>	Method to create a temporary flow instance that follows the mouse pointer for as long as the mouse button is down during flow addition
<code>void AddMaterial(void);</code>	Method to add a material flow to the model
<code>void AddEnergy(void);</code>	Method to add an energy flow to the model
<code>void AddSignal(void);</code>	Method to add a signal flow to the model
<code>// Model editing methods</code>	
<code>void MoveConnect();</code>	Method to move an element or reconnect



	the ends of a flow
<code>void MoveConnectDynamic();</code>	Method to update the screen with a temporary element instance for as long as the mouse button is down after selecting an element to be moved or reconnected
<code>void DetachEdgesFromElement(CElement* pElement);</code>	Method to delete the topological connections of a flow after its ends have been detached from a node
<code>// Model reasoning methods</code>	
<code>void Set_OrphanFlowMsg();</code>	Method to compose <code>Msg_OrphanFlow</code>
<code>void Set_BarrenFlowMsg();</code>	Method to compose <code>Msg_BarrenFlow</code>
<code>void Set_OneInManyOutMsg_M();</code>	Method to compose <code>Msg_OneInManyOut_M</code>
<code>void Set_OneInManyOutMsg_E();</code>	Method to compose <code>Msg_OneInManyOut_E</code>
<code>void Set_ManyInOneOutMsg_M();</code>	Method to compose <code>Msg_ManyInOneOut_M</code>
<code>void Set_ManyInOneOutMsg_E();</code>	Method to compose <code>Msg_ManyInOneOut_E</code>
<code>void Set_ManyInManyOutMsg();</code>	Method to compose <code>Msg_ManyInManyOut</code>

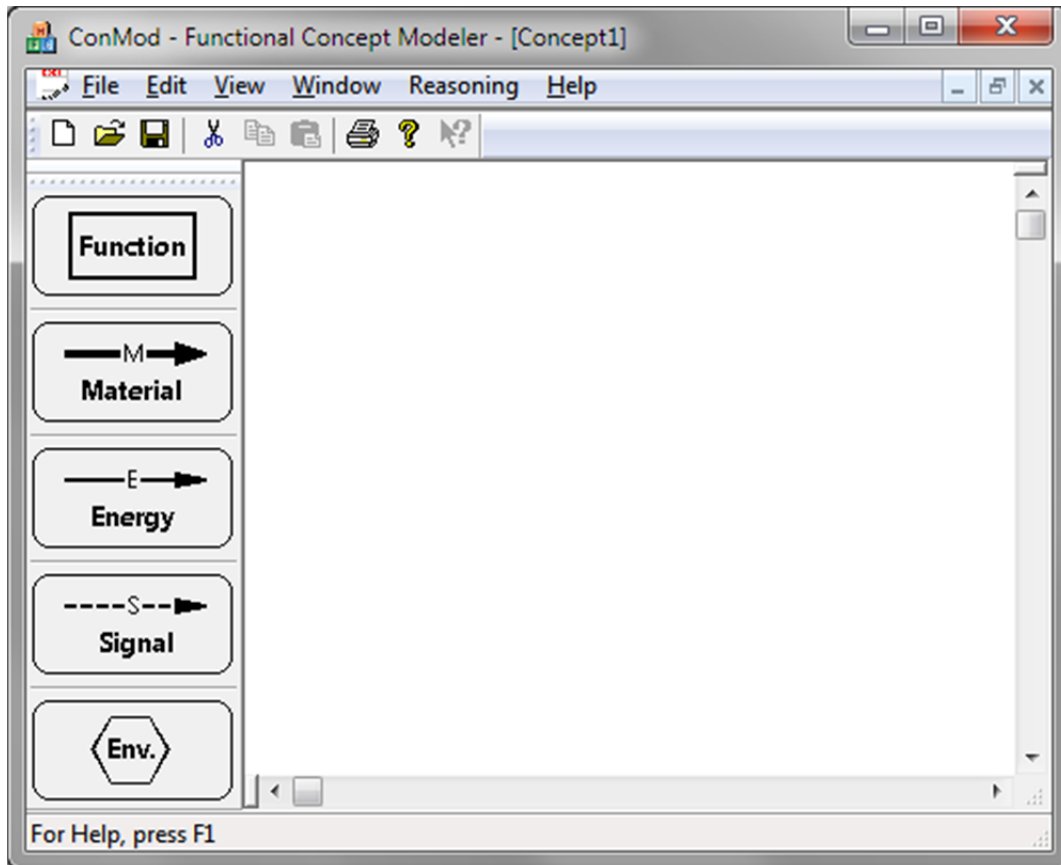
A few differences can be noticed between the implementation of these classes and the design presented in Table 5.2 and Figure 5.1, which need to be explained. The

GivenName attribute included in the CElement class is overridden in the leaf-level classes such as Material, Energy, and Signal, as the CDialog class requires the GivenName data member to be declared at the leaf classes. The HeadNode and TailNode pointers are declared in the high-level class CElement as CElement\* pointers, instead of in the CEdge class as CNode\* pointers. This change at the implementation level provides some flexibility to refer to a flow as an element, without having to know its type. The carrier-carried relation and the actions of adding a carried flow to its carrier are implemented indirectly using two actions: (1) by assigning the carrier flow as the pHeadElem of the carried flow and (2) setting the Boolean parameter ThisFlowIsOutgoingBaggage to TRUE. In addition, almost every class includes data and methods necessary to render, select, highlight, or edit their instances, which are details pertinent to this implementation rather than the design of the representation. If the representation was implemented in another application for a different purpose, these additional data and methods would probably be implemented differently, while the core class hierarchy and the data and methods prescribed in the design would not. Besides these minor implementation-specific changes, the data and methods in these classes do reflect the attributes and relations identified in Chapter 5. The next section presents the user interface design of the ConMod application.

### ConMod Graphic User Interface and Rendering

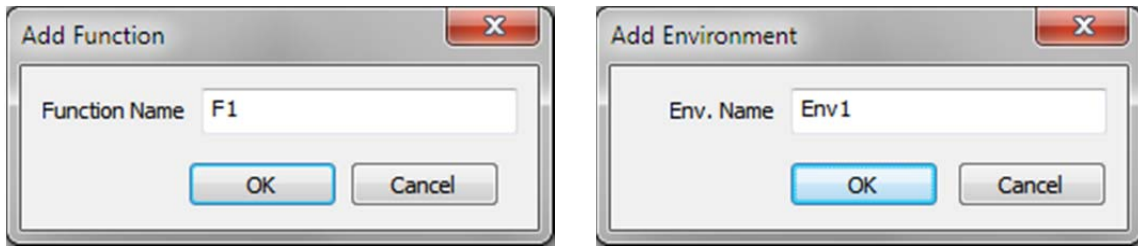
Figure 6.12 shows the design of the main window and toolbar for the application. The toolbar includes icons for the two node subtypes—function and environment—and the three flow subtypes—material, energy, and signal. The environment subtypes, source

and sink, are not implemented in this version of ConMod and can be included in a future extension.



**Figure 6.12: ConMod main window and toolbar buttons (Layer One)**

Figure 6.13 shows the Add Function and Add Environment dialogs, which pass the user-entered string to the GivenName data member of the respective classes: CFunction and CEnv. The GeometricCenter data required by those class constructors is passed from the mouse click on the graphics area of the application.

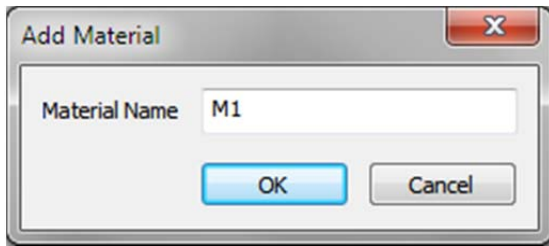


**Add Function dialog**

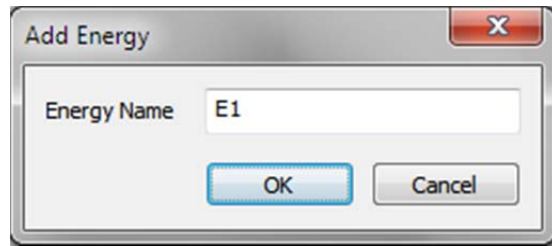
**Add Environment dialog**

**Figure 6.13: Dialog boxes for adding function and environment instances (Layer One)**

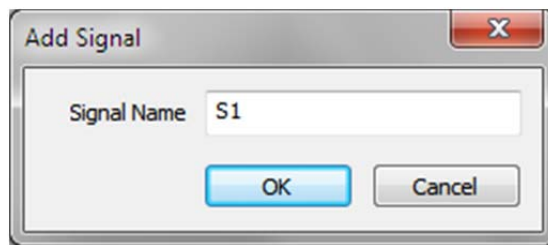
Figure 6.14 shows the Add Material, Add Energy, and Add Signal dialogs. For the purpose of the first layer of the representation, these dialogs operate in similar manner as the Add Function dialog, as they pass the user-entered name of the flows to GivenName data members in the respective classes: CMaterial, CEnergy, and CSignal. The start and end points of the flows are passed by the mouse button press and release positions on the graphic screen using two methods within the CConModView class: OnMouseDown and OnMouseLUp.



**Add Material dialog**



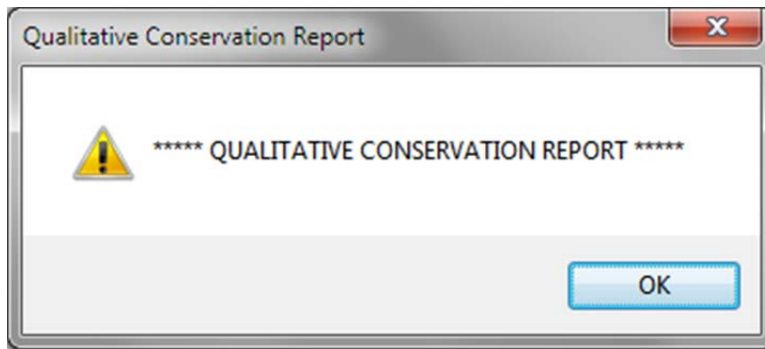
**Add Energy dialog**



**Add Signal Dialog**

**Figure 6.14: Dialog boxes for adding Material, Energy, and Signal instances (Layer One)**

The model entities such as functions, flows, and environments can be added, edited, and deleted using this interface objects, as illustrated next. The reasoning algorithms can be invoked by double-clicking on the white space in the graphics area, while a message window is provided for returning reasoning output. When run on an empty model, this window returns a default hard-coded message, shown in Figure 6.15.


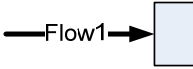
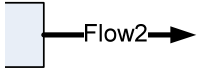
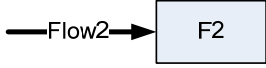
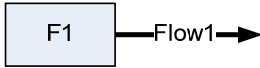
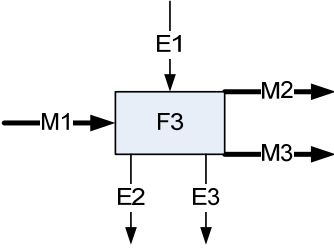
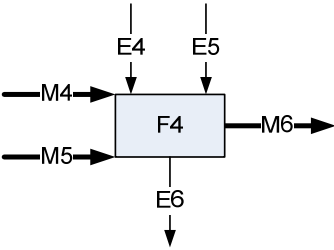


**Figure 6.15: Qualitative conservation reasoning message dialog**

### **6.2.2 Demonstration of Function Modeling and Qualitative Conservation Reasoning**

As mentioned in Chapter 5, the conservation layer of the representation is designed to support the first eight out of the twelve reasoning tasks summarized in Table 4.15. The ConMod application is used here to demonstrate this reasoning. The graphical constructs that trigger these reasoning and the message to be generated from these reasoning are summarized in Table 6.14. In addition to the eight types, a ninth reasoning—many-in-many-out—is identified by logically extending the reasoning types of Table 4.15. In this reasoning (# 9a in Table 6.14), the system identifies functions that input and output multiple flows of subtype Material or Energy and accordingly infers conservation.

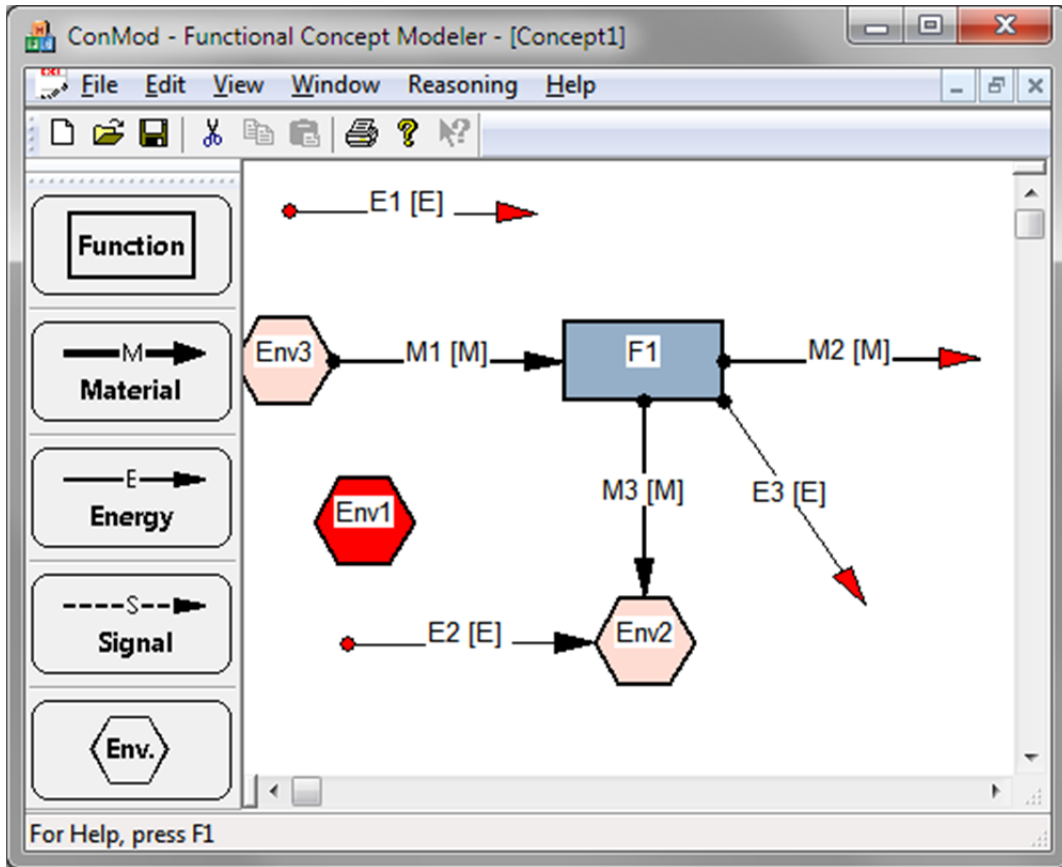
**Table 6.14: Conservation reasoning to be validated using ConMod**

Ref. from Table 4.15	Reasoning Name	Model Construct	Expected Feedback
<b>Topologic</b>	1		Redundant function: "F1"
	2		Dangling tail: Flow1
	3		Dangling head: Flow2
<b>Derivational</b>	4		Barren flow: Flow2
	5		Orphan flow: Flow1
	6		Conservation inferred: $\{M1\} \rightarrow \{M2, M3\}$ $\{E1\} \rightarrow \{E2, E3\}$
	7		Conservation inferred: $\{M5, M5\} \rightarrow \{M6\}$ $\{E5, E5\} \rightarrow \{E6\}$

Ref. from Table 4.15		Reasoning Name	Model Construct	Expected Feedback
	9	Material transformation without energy exchange		Material transformation without energy in F7
	9a	Many-in-many-out inference		Conservation inferred: $\{M7, M8\} \rightarrow \{M9, M10\}$

Topologic and derivational reasoning are demonstrated by constructing the model in Figure 6.16. The model shows at least one instance of all five element classes shown on the toolbar. The flow subtypes Material and Energy are indicated by their line thickness (material = thick, energy = thin). Signals have a dotted line font and S1 is carried by Flow2. Element names such as M1 and F1 are entered by the user through the text field in the respective dialogs. The letters within brackets on flow names are classes selected from the tree in those dialogs. As seen in Figure 6.14, the default selections for the flow subtypes are M for Material and E for Energy. These trees could be expanded to select a more specific flow type, but that detail is outside the scope of this validation of Layer 1.

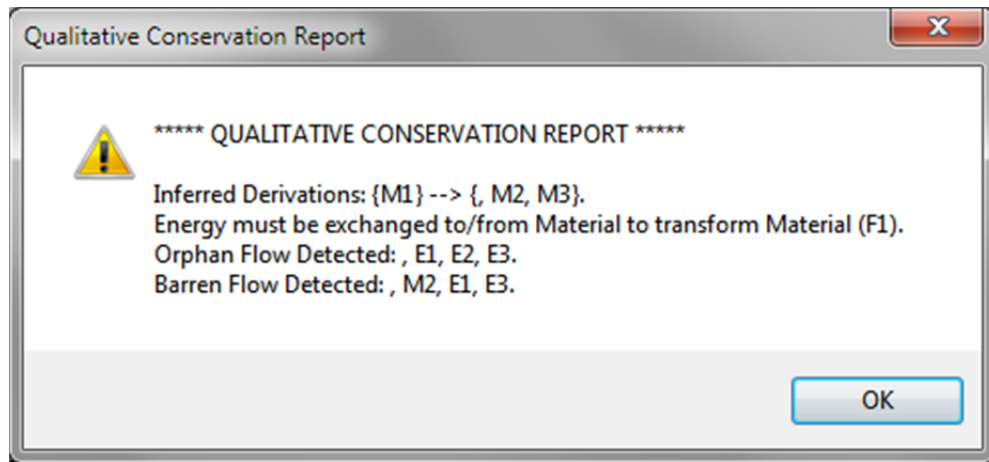




**Figure 6.16: Model for topologic reasoning and derivational reasoning (# 1 – 5)**

**Topological Reasoning** is performed at real-time during modeling, as indicated by the colors of the elements. F2 and Env1 are identified as redundant (red), as they are not attached to any flow. F1, Env2, and Env3 are not redundant, as they have at least one attached flow. The dangling flow ends are highlighted in red, while attached ends are black. For example, E2 has dangling tail but attached head, M2, E3, and S1 have dangling heads but attached tails, E1 has both ends dangling, while M1 and M3 have both ends attached to nodes. **Thus, reasoning # 1, # 2, and # 3 from Table 6.14 are demonstrated.**

**Derivational Reasoning** is performed on user-request, by double clicking the white space in the graphics area. The result of derivational reasoning on the model is shown in Figure 6.17. All checks are run at every request. In this case, the reasoner found E1, E2, and E3 to be orphan, since their parent flows are not modeled. This, despite the fact that E1 and E2 have dangling tails but E3's tail is attached. The derivational inference is not based upon dangling end status – they are based upon whether the parent and child flows are modeled or not. M2, E1, and E3 are barren, since their children are not shown. Although the children of M3 and E2 are not shown, they are not detected as barren, since they are released to an environment, and thus their children are outside the scope of reasoning. Similarly, M1 is not an orphan, since its parent flows are outside the reasoning scope. **Thus, reasoning # 4 and # 5 from Table 6.14 are demonstrated.**



**Figure 6.17: Derivational reasoning output from the model in Figure 6.16**

In addition, this model demonstrates the one-in-many-out inference for material flows. M1 is inferred to be the parent of M2 and M3, as M1 is the only Material input to

F1 while M2 and M3 are the only Material output. **Thus, reasoning # 6 is partially demonstrated.**

Messages for the other reasoning are not produced, since the model either passed those checks or did not include modeling constructs where they apply. For example, the many-in-one-out or many-in-many-out constructs are not modeled. In order to demonstrate reasoning # 6 through 9a, the model is edited to Figure 6.18. The reasoning output is shown in Figure 6.19. The first two messages are results of many-in-one-out derivation inferences for Material and Energy. The third message indicates that F1 transforms material without consuming or releasing energy, which is a violation of the first law of thermodynamics. Notably, flows with inferred children (e.g., M1) are not listed as barren and flows with inferred parents (e.g., E3) are not identified as orphans. **With this illustration, reasoning # 6, # 7, and # 9 are demonstrated.**

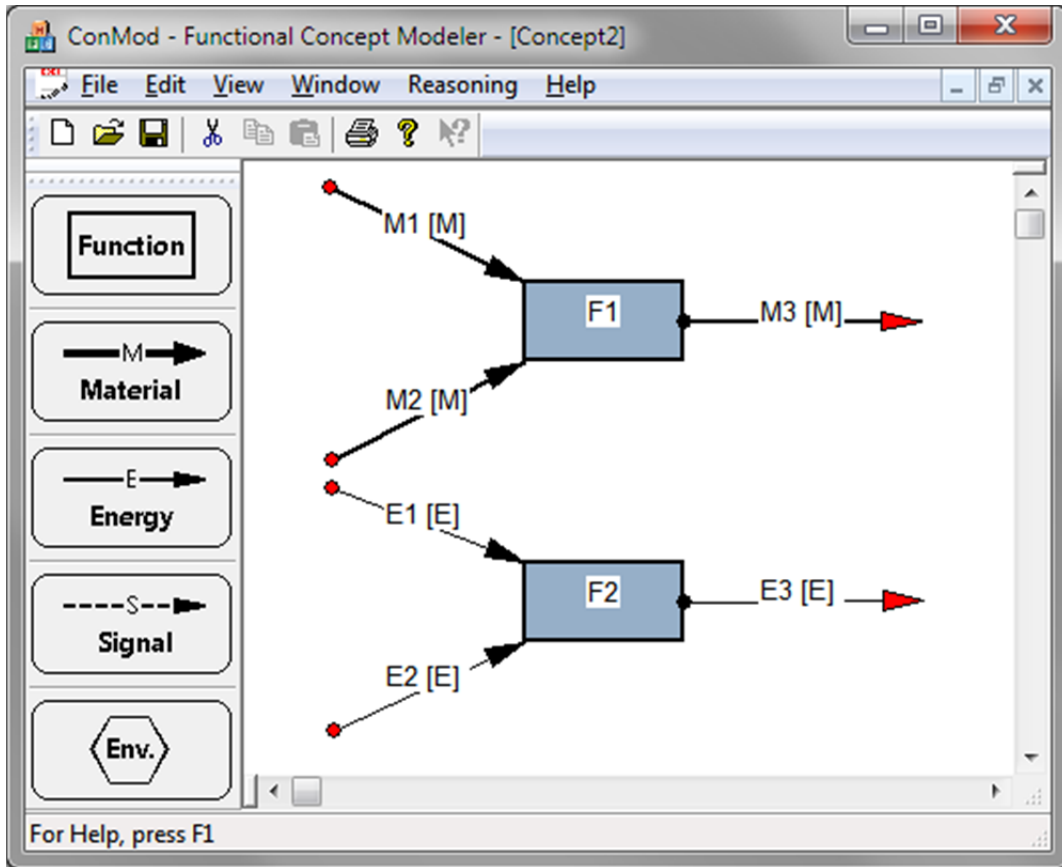
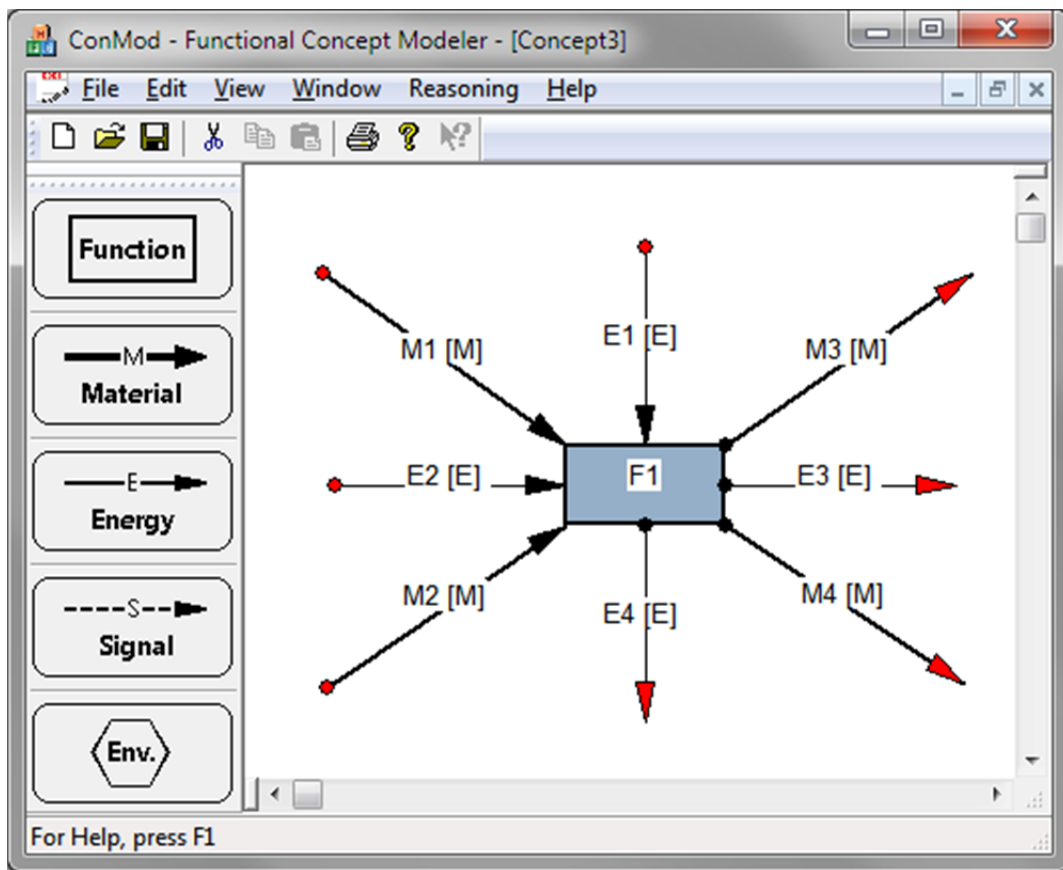


Figure 6.18: Model for derivational reasoning # 6, 7, and 9

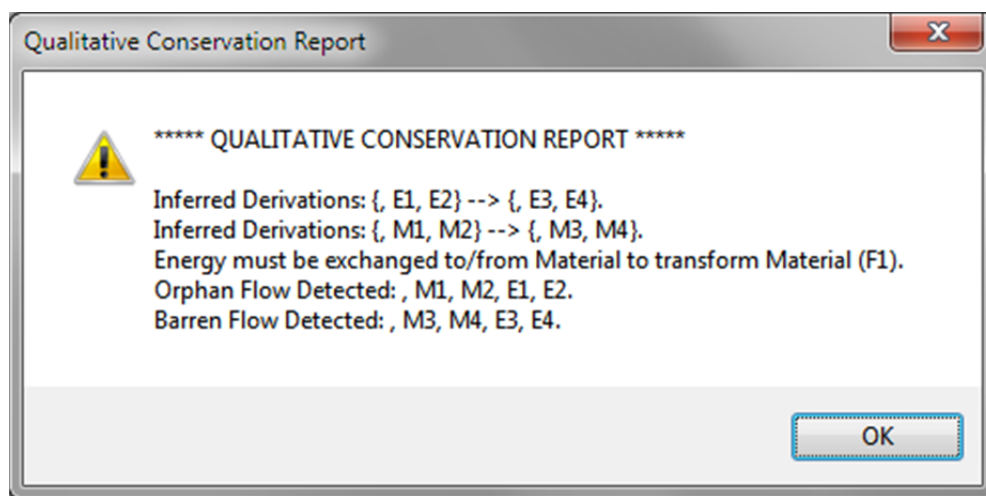


Figure 6.19: Derivational reasoning output from the model in Figure 6.18

To demonstrate the last reasoning, many-in-many-out inference (# 9a), the model in Figure 6.20 is constructed. The reasoning output is shown in Figure 6.21. Each flow in the left side of a derivation relation ( $\rightarrow$ ) is a parent of each of the flows on the right side of the relation. Thus, E3 is a child of both E1 and E2, while the same is true for E4. Thus, the individual conservation relations are not captured in this inference. Similar to the previous models, the orphan and barren flow messages report flows, whose parent of children cannot be inferred from the model. **Thus, reasoning # 9a is demonstrated.**



**Figure 6.20: Model for derivational reasoning # 9a**



**Figure 6.21: Derivational reasoning output from the model in Figure 6.20**

The demonstration of the nine reasoning tasks (eight from Table 4.15, plus #9a in Table 6.14) using the ConMod software illustrates that the representation can be incorporated in software to support conservational reasoning computationally. **Thus, these demonstrations constitute external validation of the representation against the laws of conservation.** While the reasoning algorithms are illustrated using models of small complexity that are limited to one or two functions, it remains to be tested if the representation can support constructing models of larger scales and support reasoning on them, which is addressed next.

### **6.2.3 Application to Product-Level Modeling and Reasoning (Scalability)**

To test scalability of the representation (Section 3.6), the air heater function structure from Model State 4.14 is constructed using ConMod. The model has eight functions and twenty flows, and is thus considered a mid-size model, based on the

distribution of model sizes in the Design Repository [166]. Thus, this model constitutes a low-fidelity demonstration of scaling.

The model is shown in Figure 6.22. Although Model State 4.14 shows residual flows distinctly with red arrows, no flow in this model is tagged as a residual, since the depiction of residues is not included in this layer of the representation. Residual flows are reserved for the second layer, presented in Chapter 7. However, this model shows the use of carrier flows. For example, KE1 is added to the carrier Air3 by function En\_Air1. En\_Air1 shows the addition of kinetic energy using mechanical work MW1 (e.g., in a fan blade), which is available by converting EE1 (e.g., in a motor). En\_Air3 adds heat ThE1 to the air flow, part of which (ThE2) is lost as Loss9 to the environment. The Transfer Air function consumes some kinetic energy from the flow itself, which is used to overcome the frictional resistance of the flow path and lost as Loss8. Notably, Model State 4.14, being constructed by a human designer prior to the development of this representation, contains a violation of Rule 31 (see Construct 74), as it shows the carried flow ThE2 being input to a node that does not input its carrier, Air4. This deviation is addressed in the model of Figure 6.22 by including one additional function, Dissipate, which shows the loss of heat from the exiting air stream through possibly the walls of the surrounding conduit.

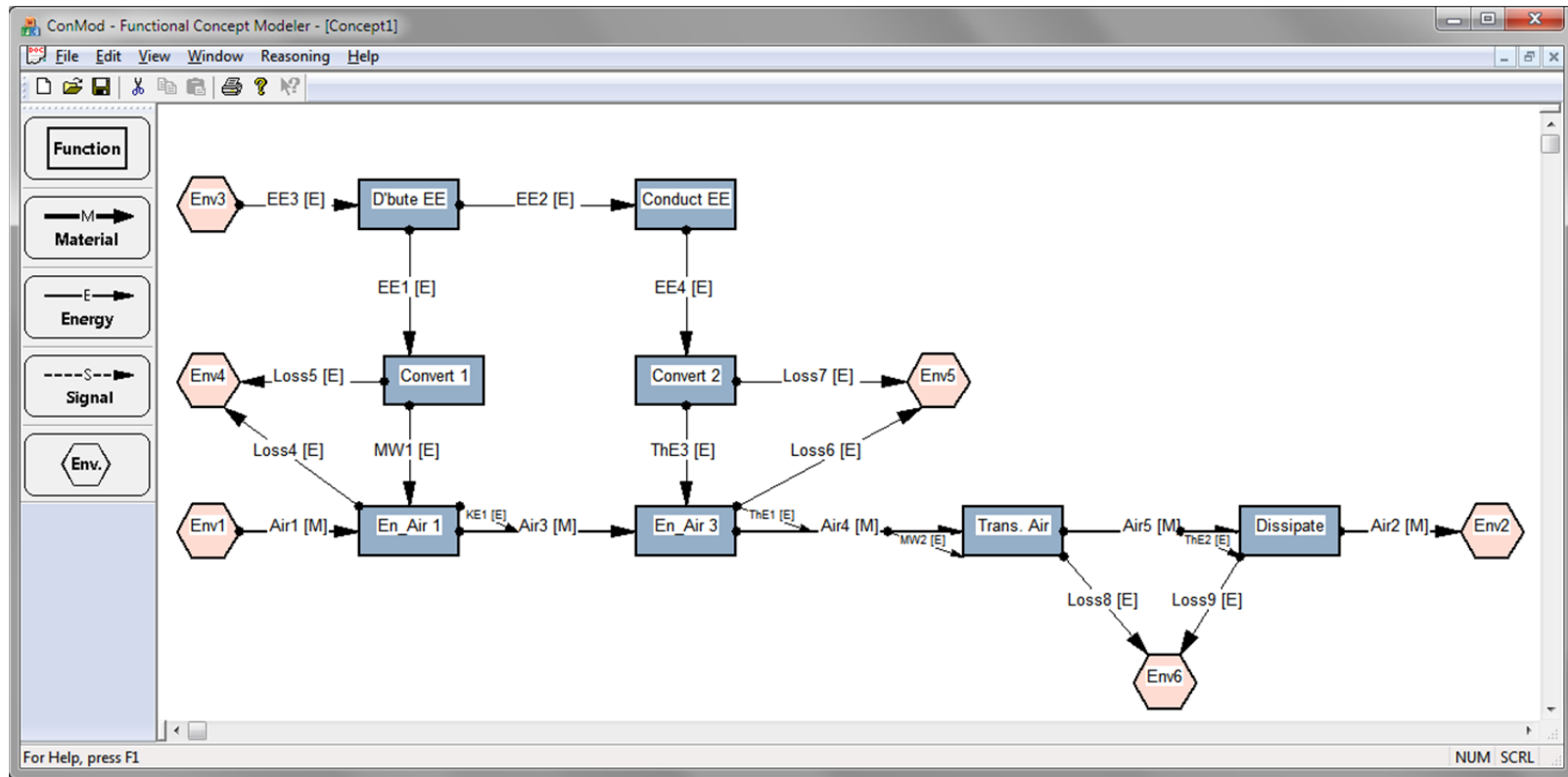
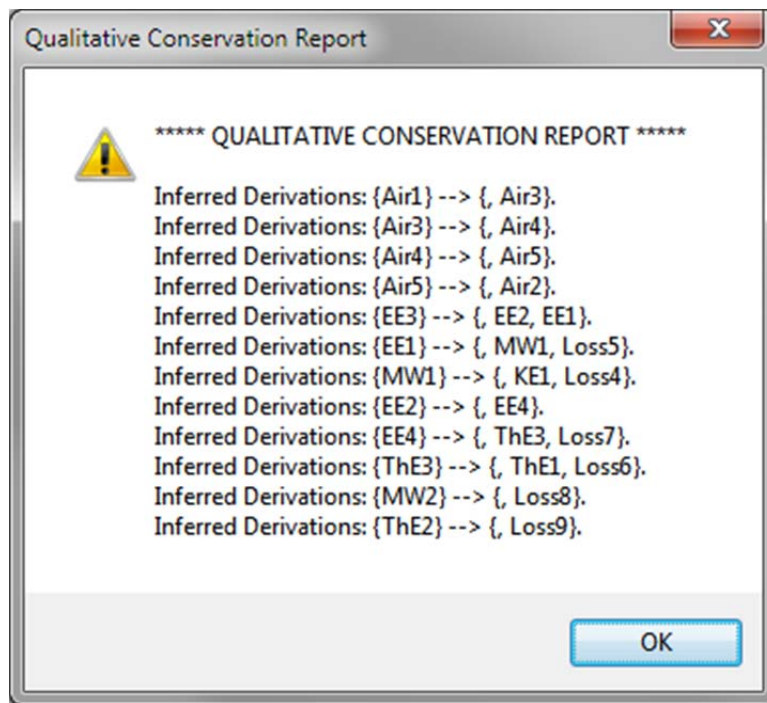


Figure 6.22: The Air Heater model from Model State 4.14 reconstructed in ConMod



Figure 6.23 shows the reasoning output from this model. None of the flows is identified as orphan or barren, as each flow has either a node or a carrier flow at its head or tail. Each function that transforms a material flow either adds or extracts energy to or from at least one of the material flows. Finally, the derivations are shown as inferred, based on the count of input and output flow of the two major subtypes—Material and Energy—attached to each function. **With this illustration of product modeling and reasoning using ConMod, the scalability of the representation is demonstrated.**



**Figure 6.23: Reasoning output for the Air Heater model in Model State 4.14**

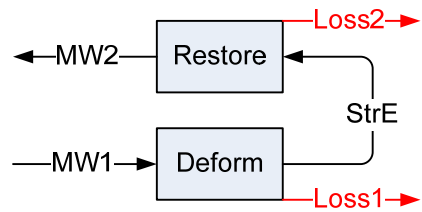
In conclusion, this chapter provides validation for the first layer of the representation, including internal consistency of the vocabulary and the grammar, and external validation using modeling and reasoning through software implementation.

Internal consistency is established through logical examination of the exhaustiveness and consistency of the grammar, and by ontological examination of consistency of the vocabulary. External validation is claimed by implementing the representation in the ConMod software and using it to construct function structure models and perform two types of conservation reasoning: topologic and derivational. **Based on this discussion, it is asserted that the first layer of the representation is validated.**

Next, the representation is extended to support qualitative and quantitative reasoning based on the principle of irreversibility. While conservation of energy, supported in the first layer qualitatively, is a corollary of the first law of thermodynamics [128, 129], the principle of irreversibility is a consequence of the second law of thermodynamics [128, 129]. Thus, after the extension presented next, the representation will stand validated against both laws of thermodynamics.

## CHAPTER 7. REPRESENTATION LAYER TWO: EXTENSION OF LAYER ONE FOR IRREVERSIBILITY-BASED REASONING

The formal representation presented in the previous chapter is extended in this chapter to support one additional type of reasoning—irreversibility—identified in Table 4.15. The extension is needed in order to support validating design concepts against the principle of irreversibility. At the qualitative level, this reasoning includes the detection of omitted residual energy flows in functions that input and output energy, based directly on the second law of thermodynamics that requires any physical process—and thus, any function—to be irreversible. For example, mechanical work required to deform an elastic member such as a spring is partially lost as heat during deformation, thus requiring more work to be input to the spring than the amount of strain energy stored. The work done in producing this heat cannot be recovered during elastic recovery of the spring. Instead, during recovery, part of the stored energy is again lost in overcoming internal friction of the material and wasted as heat, thus further reducing the available work output. Thus, energy is lost in both steps: during storing strain energy in the spring and recovering mechanical work from it, thus making the cyclic process irreversible. This loss of energy is shown in Figure 7.1, where MW is mechanical work and StrE is strain energy.



**Figure 7.1: Energy loss in physical processes**

This irreversibility in cyclic processes is a consequence of the second law of thermodynamics, a fundamental law that no design principle or concept can escape. One corollary of this law is that when a system is put through a process that takes it from one state to another, it is impossible to completely reverse the process such that both the system and its surroundings are put back to their previous states [128, 129]. All real processes are irreversible. In this example, the spring can return to its original thermodynamic state (undeformed state) at the end of each cycle, as long as the deformation was within the elastic limit, the recovery was complete, and time was allowed for the spring to exchange heat with the environment, but the environment around the spring undergoes change of state in every cycle, as it gains heat dissipated by the spring.

This residual energy often becomes major design consideration. For example, energy rejected by the spring or any elastic medium initiates the study of hysteretic losses, a major concern in the design of tires and other traction systems [176]. Similarly, heat rejected by internal combustion engines is a residual from the combustion process that necessitates the entire cooling subsystem in automotive and other applications in order to be dissipated to the atmosphere. Since this irreversibility is known to be

inviolable by all design concepts and can become a major design concern, it is deemed useful to capture this principle in the formal representation, so that concepts can be validated against this principle and designers can be alerted about losses their design must incur, thus preventing accidental overestimation of output power or underestimation of power required to operate a device. In the representation, irreversibility is first captured qualitatively so that algorithms can detect missing residual flows. Further, the extendibility to support quantitative reasoning is illustrated by adding new function and flow attributes, and using them to algorithmically compute the quantitative efficiency of individual functions and the model as a whole. Section 7.1 presents the extension to the representation. Sections 7.2 and 7.3 demonstrate irreversibility-based reasoning of qualitative and quantitative types using extension of ConMod, thus providing external validation of the extension.

### 7.1 Extension of the Representation to Include Irreversibility-Based Attributes

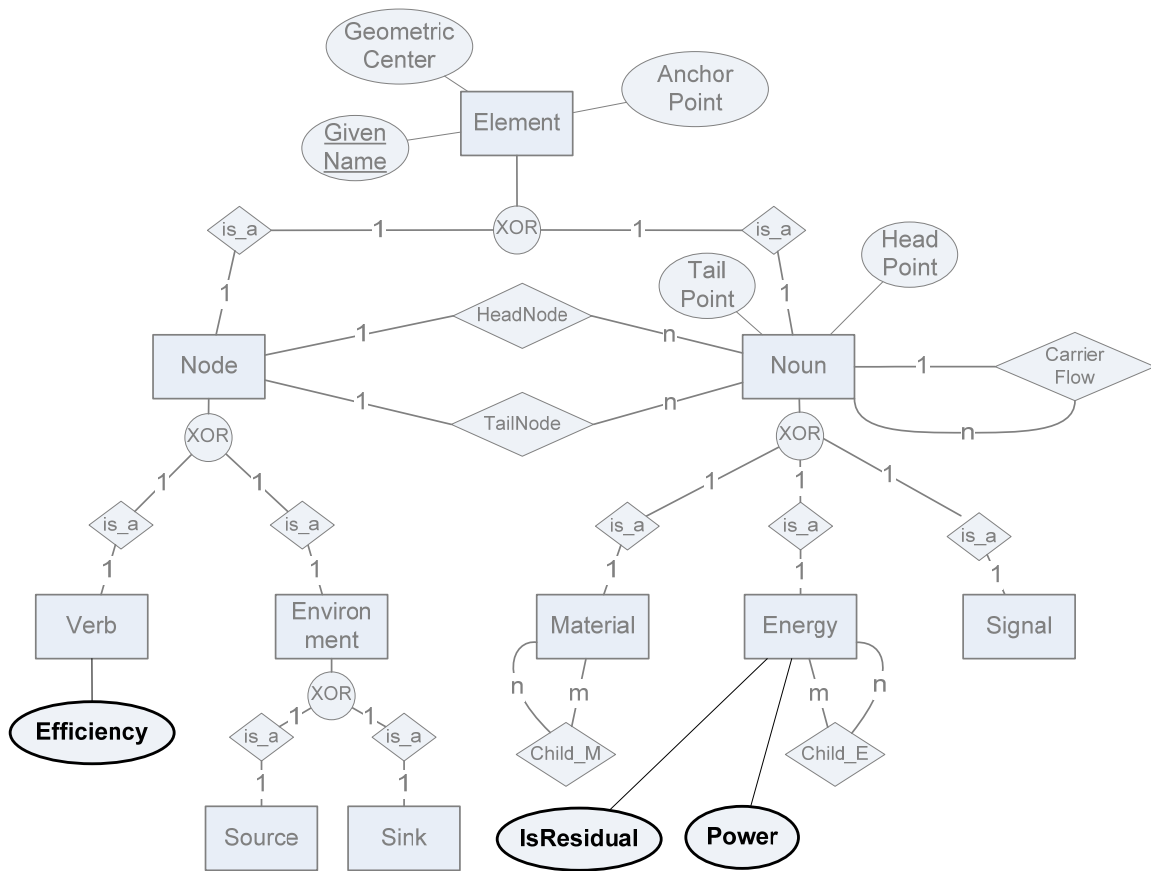
The extension of the representation to support irreversibility reasoning requires adding only one attribute (IsResidual) for quantitative reasoning and two other attributes (Efficiency and Power) for quantitative reasoning. Table 7.1 describes these attributes, their classes, and data types.

**Table 7.1: Layer 2 attributes and descriptions**

<b>Attribute</b>	<b>In Class</b>	<b>Data Type</b>	<b>Description</b>
IsResidual	Energy	Boolean	IsResidual is set to true when a designer declares that an energy flow included in the model is

<b>Attribute</b>	<b>In Class</b>	<b>Data Type</b>	<b>Description</b>
			residual from a function.
Power	Energy	Number (e.g., float or double)	The attribute Power of an Energy flow is the time rate of energy carried by that flow instance. It is predicated on the representation being used to model steady-state flow processes of mechanical systems. Many devices, process, and principles can be modeled as such, while the representation can be later extended to model transient or non-flow processes in the future.
Efficiency	Verb	Number (e.g., float or double)	Efficiency is a number variable to hold the efficiency of a function as the ratio of the total power of the output energy flows not marked as residual to the total power of all input flows to the function.

Figure 7.2 shows the extension of the representation, using the ERA model from the first layer (Figure 5.1) in grey and the newly added attributes in bold black font. The following sections illustrate software implementation of this extension and validation by supporting both qualitative and quantitative reasoning based on the irreversibility principle, using ConMod.



**Figure 7.2: Extension of the representation to support irreversibility reasoning**

## 7.2 Implementation and Validation: Qualitative Irreversibility Reasoning

Since the extension is based on the first layer, whose internal consistency is already established (Section 6.1), it is unnecessary to re-examine the internal consistency of the representation post-extension. The attributes added in this extension are mutually disjoint, as (1) efficiency applies to the Verb class while the other two (IsResidual and Power) apply to its disjoint class: Noun, and (2) the two attributes added to Noun are independent of each other, since all Energy flows must have the Power attribute independent of whether it is declared by the modeler as residual or not. Further, the new attributes do not require any grammar rule to restrict their application: all functions have

efficiency (whether declared by the user or inferred by reasoning), all Energy flows have Power (declared or computed), and all Energy flows have a value for the IsResidual attribute (true or false). Thus, by this extension, the internal consistency of the representation could not have altered from that established in the previous layer. However, the external validity should be demonstrated using illustration of reasoning, as shown next.

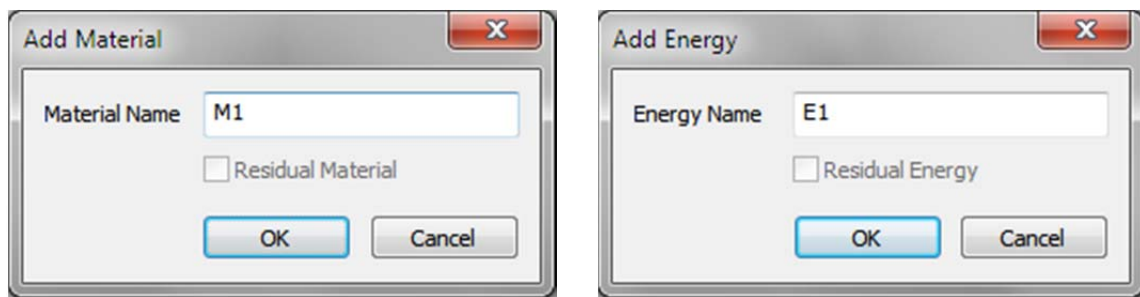
There is no change to the class diagram of ConMod due to this extension, as the extension is limited to addition of attributes to existing classes only. The following changes are made to the user interface:

1. A Residual Energy check box is added to the Add Energy dialog (Figure 7.3a)
2. A Residual Material check box is added to the Add Material dialog (Figure 7.3b)
3. A menu is added to turn on irreversibility-based modeling and reasoning (Figure 7.4)

The Residual Energy check box passes a Boolean value to the IsResidual attribute of the CEnergy class (checked = true, unchecked = false) and is used in the qualitative and quantitative irreversibility reasoning. The Residual Material check box passes values to the CMaterial class, but is not used in reasoning in this implementation of ConMod, and hence is not mentioned as a necessary data member in the previous section. Residual Material is included to provide the ability to mark a material flow as residual for visualization and human reasoning. Both check boxes are turned on when irreversibility-



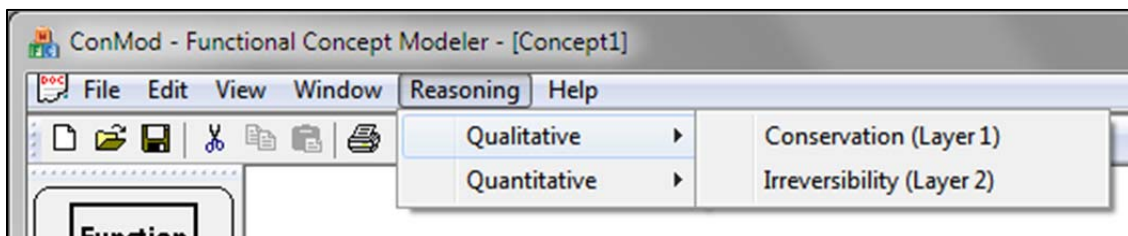
based modeling and reasoning is selected from the Reasoning menu. In the Reasoning menu, selecting Reasoning > Qualitative > Conservation (Layer 1) limits the reasoning ability to that provided by the previous implementation of ConMod (hereafter called ConMod-1). Irreversibility extension reasoning can be turned on by selecting Reasoning > Qualitative > Irreversibility (Layer 2).



(a) Add Material dialog

(b) Add Energy dialog

**Figure 7.3: Dialog boxes for adding Material and Energy instances (Layer Two - Qualitative)**



**Figure 7.4: Reasoning menu options in ConMod (Layer Two - Qualitative)**

To demonstrate irreversibility reasoning at the qualitative level, the Air Heater model of Figure 6.22 is reconstructed using the Layer Two implementation of ConMod (hereafter referred as ConMod-2), shown in Figure 7.5. Although some flows are named

as Loss6 and Loss7, none are marked as Residual in in this model, in order to maintain similarity with Figure 6.22 constructed using ConMod-1. The derivational inferences produced by ConMod-2 (Figure 7.6) are identical as that produced by ConMod-1 (Figure 6.23). An inspection of the model and this report reveals that the inferences are correct to the model.

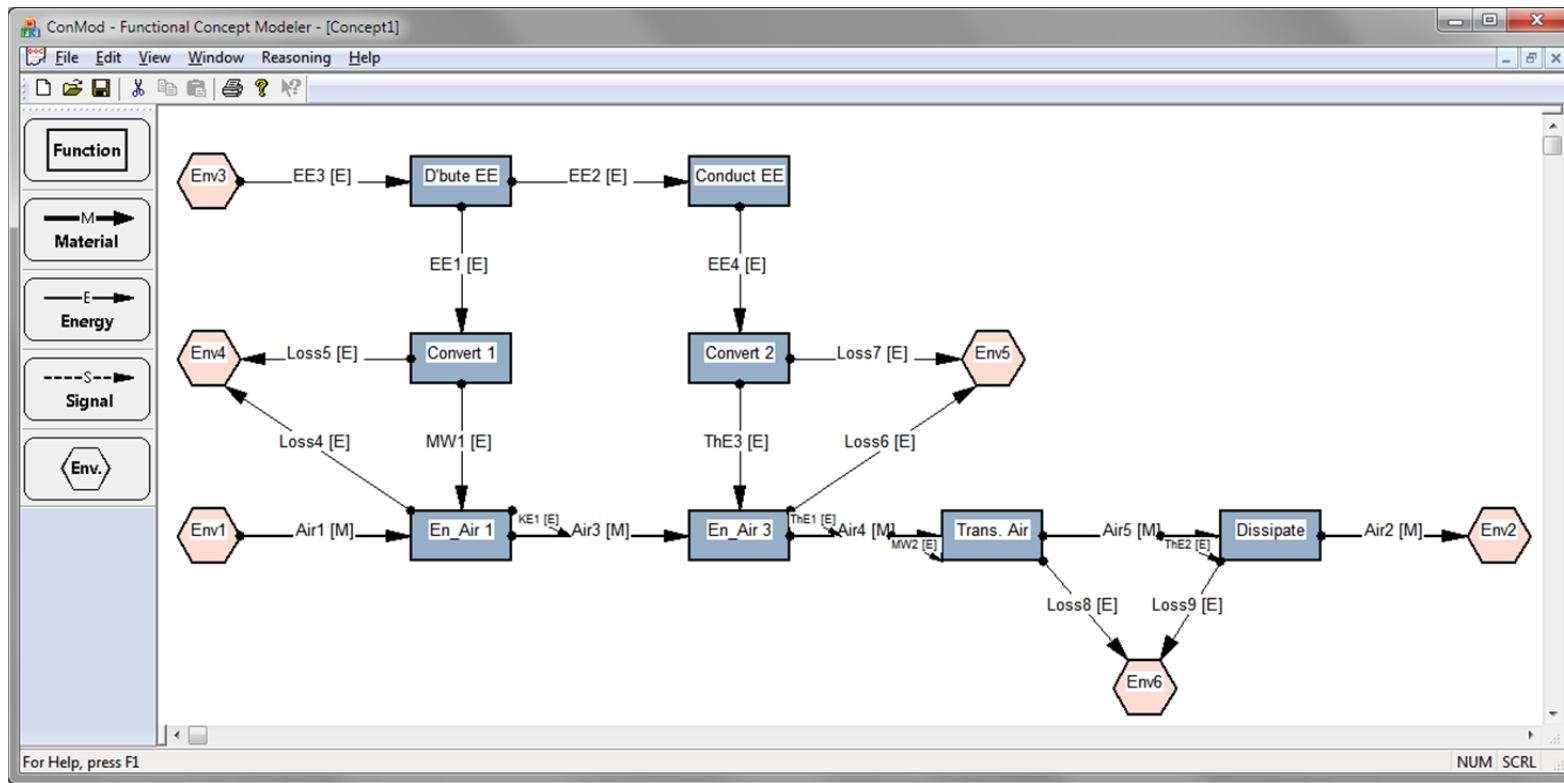
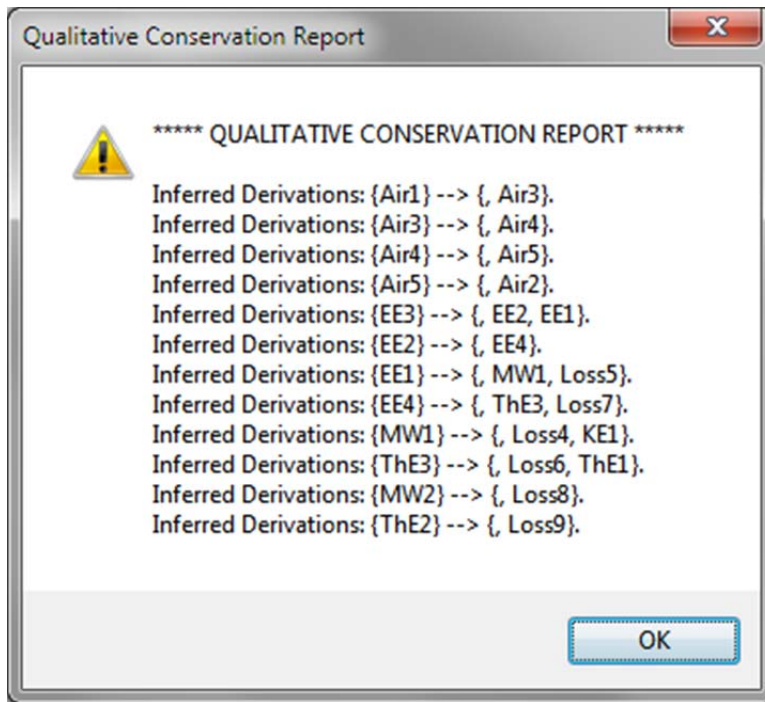
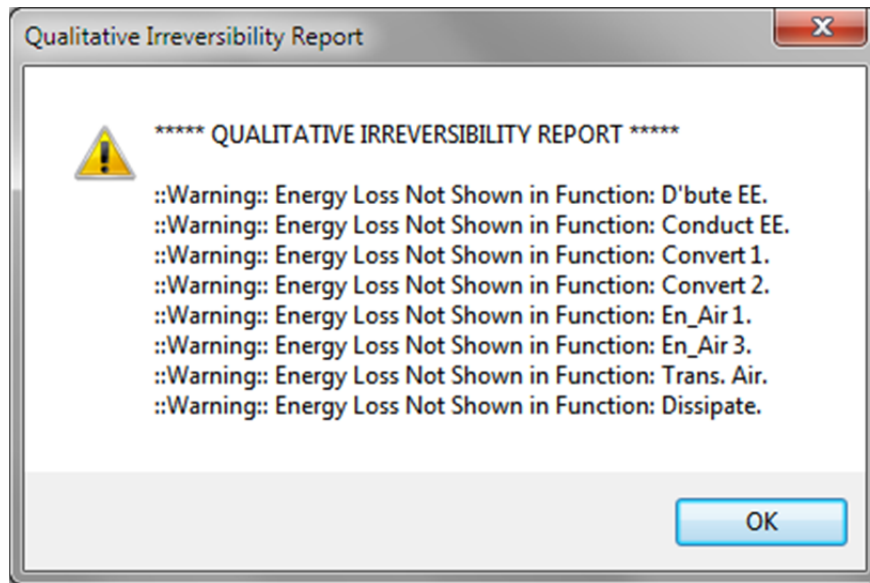


Figure 7.5: Air Heater model of Model State 4.14 reconstructed using ConMod (Layer 2)



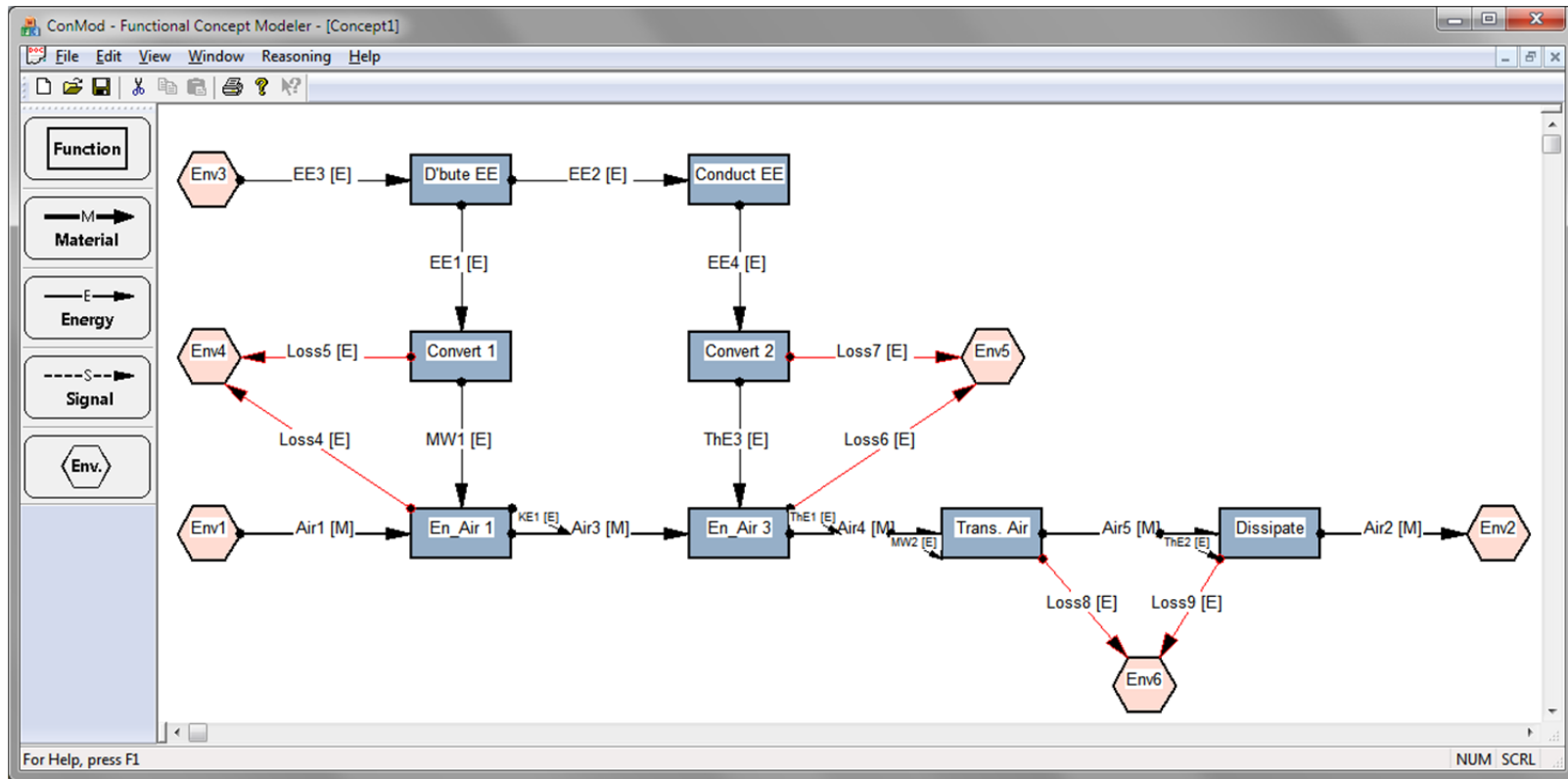
**Figure 7.6: Qualitative derivational reasoning produced by ConMod-2 on the model shown in Figure 7.5**

Further, ConMod-2 produces a report identifying functions that violate the principle of irreversibility. At a qualitative level, the only reasoning possible is to detect functions that input and output energy flows but do not have any of the output energy marked as residual. This reasoning output is shown in Figure 7.7, which identifies each function in Figure 7.5 as a violation, since none of the energy flows is marked as residual.



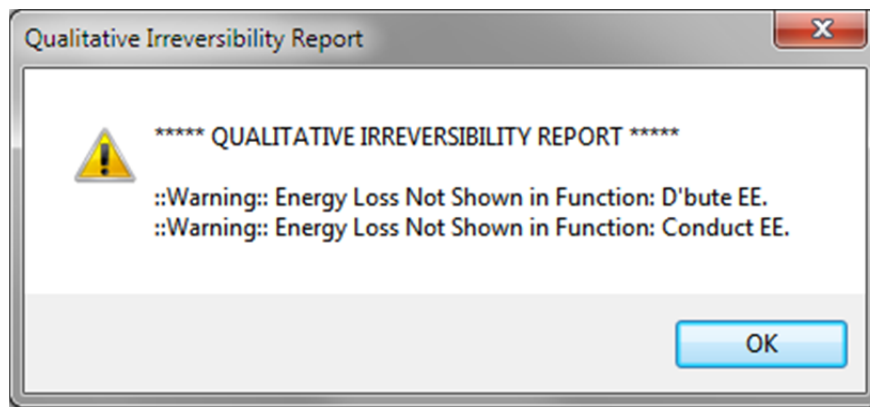
**Figure 7.7: Qualitative irreversibility report produced by ConMod-2 on the model shown in Figure 7.5**

To address these messages, the model is next edited to mark the residual flows of Model State 4.14 as residual. These flows include Loss4, Loss5, Loss6, Loss7, Loss8, and Loss9. The edited model is shown in Figure 7.8, where the residual flows are in red.



**Figure 7.8: Modified model of the Air Heater, with residual flows identified by modeler**

The derivation reasoning output from this edited model is identical to the previous model, as the tagging of an energy flow as residual should not change the inferences about total energy balance of a function. However, the qualitative irreversibility reasoning output is changed due to this edit, as shown in Figure 7.9. The two functions that continue to violate irreversibility are “D’bute EE” and “Conduct EE”.



**Figure 7.9: Qualitative irreversibility report produced by ConMod-2 on the model shown in Figure 7.8 (modified model with some residual flows marked)**

Notably, these two functions (D’bute EE and Conduct EE) were not identified to have any residual flows in Model State 4.14, produced by a human designer during the modeling exercise. However, both functions are expected to incur losses in the physical embodiments, as the distribution of electrical energy in a junction box or the conduction of electrical energy through a wire certainly produces resistive heat, and possibly other forms of energy losses. This detection of violations of the second law of thermodynamics is an illustration of how the representation and its implementation in ConMod-2 can help to draw a designer’s attention to constructs that inadvertently inflict

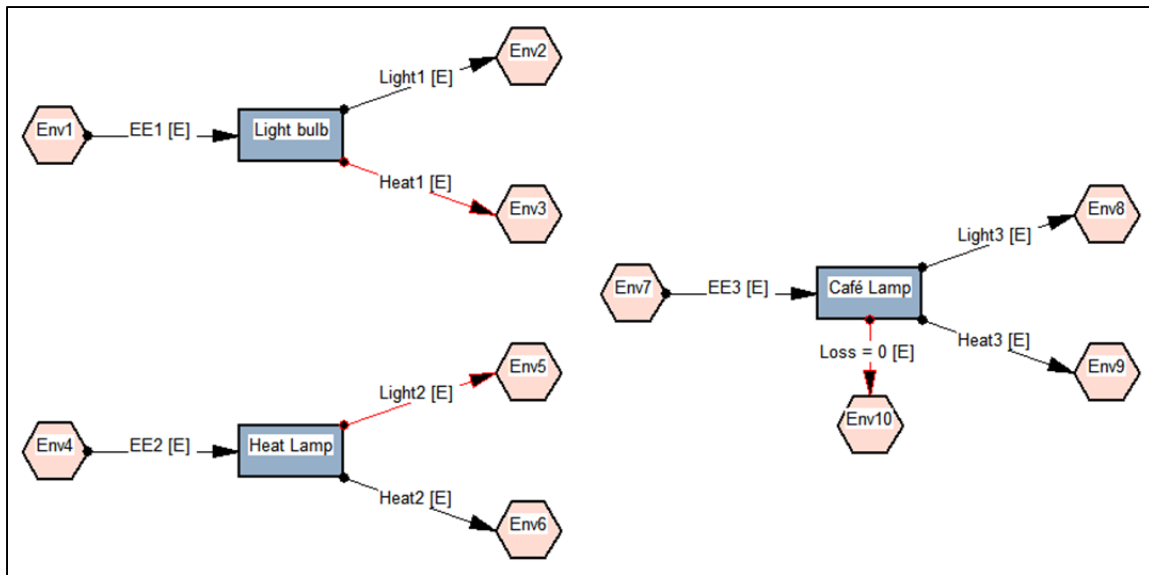
violations of physical laws in a concept. In many cases, this oversight can make a concept unrealizable if the omitted losses are of significant magnitude.

The omission of residues is detected as a warning, rather than an error of the model, as visible in Figure 7.9. This choice is made during implementing ConMod-2, due to two reasons. **First**, the notion of “loss” is not fundamentally required by the second law of thermodynamics. This law implies that when heat is converted to work by a device (e.g., a heat engine), some part of the input heat cannot be converted into useful work and must be rejected [128, 129]. Thus, the law only requires multiple output energy forms, without identifying one as loss. This rejected heat from an engine is commonly described as loss, since in typical applications, the output shaft work is the flow of interest and efficiency is described as the ratio of the output work to the input heat. However, in design projects, the notion of loss often depends on the designer’s intent, the design requirements, and the behavior of the physical principles of the device. For example, heat rejected by an automotive engine is often used to satisfy two design requirements at the system level: (1) to provide heat to the passenger cabin and (2) to raise and maintain engine temperature within a range required by the operating viscosity of the lubricants. Specifically in wintery conditions, a change often happens in the physical behavior of the engine and user-intent. (1) The rate of heat loss from the engine increases due to higher temperature gradients between the engine and the surroundings. (2) The user often intends to deliver more heat to the cabin. Thus, an engine that rejects more heat is often more desirable in winter, so that it can keep the cabin warm and maintain engine temperature at the same time. Similarly, heat produced by an electric



lamp can be perceived as loss if the intent is to produce light, while it can be a useful commodity in a heat lamp, where the light is residue. In some other applications, where both the light and heat produced by a lamp are used to achieve design goals—such as lamps in cafeteria stalls used to both illuminate and keep warm the food—none of the output flows may be considered as loss. Thus, the notion of loss depends on the design problem and requiring residual flows from every function in the representation may reduce modeling coverage over some of the cases mentioned above.

**Second**, while all physical processes are subject to irreversibility, the flexibility to optionally overrule this requirement may provide modeling convenience, especially when the residual flows are negligible or of unknown magnitude. In order to allow the modeler to capture negligible or unknown residues without violating irreversibility at a qualitative level, the representation allows setting zero magnitude for residual flows at a quantitative level, while the qualitative reasoner detects all functions without residual energy as violations. Thus, all three models shown in Figure 7.10—the light bulb, the heat lamp, and the café lamp discussed above—are accepted by the reasoner at a qualitative level, where the loss from the café lamp is set to zero magnitude in the quantitative model. The quantitative extension of the representation is discussed next.



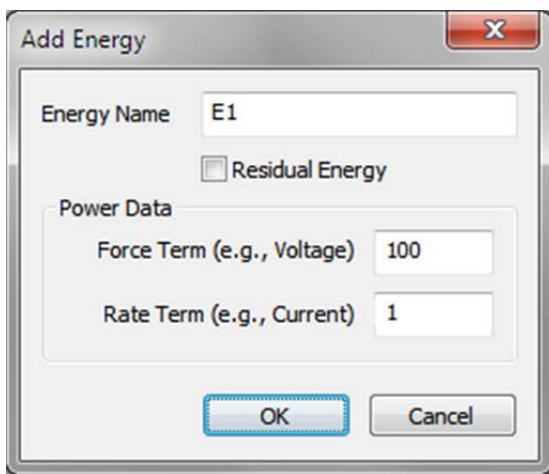
**Figure 7.10: Depiction of residual flows for different design intent and zero magnitude**

In summary, the representation is extended to include class attributes necessary for supporting qualitative irreversibility-based reasoning. The extension of ConMod-1 into ConMod-2 is used to demonstrate that such reasoning is supported by an implementation of the extension. **Thus, the extensibility of the representation to support qualitative irreversibility reasoning is validated.** Next, the representation is further extended to support quantitative reasoning.

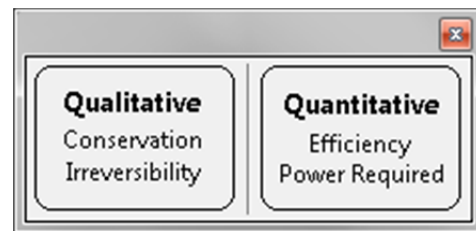
### 7.3 Implementation and Validation: Quantitative Irreversibility Reasoning

The quantitative extension is implemented in ConMod-2 by adding the Efficiency and Power attributes to the CFunction and CEnergy classes. No change is made to the class structure. The user interface is extended as follows:

1. Two number fields—Force Term and Rate Term—are added to the Add Energy dialog to capture power associated with an Energy flow instance (Figure 7.11a).
2. Two buttons are added on a new reasoning toolbar to allow requesting the qualitative and quantitative reasoning algorithms separately (Figure 7.11b).
3. A menu is added to turn on quantitative modeling and reasoning (Figure 7.12).



(a) Add Energy dialog



(b) Reasoning toolbar

Figure 7.11: Add Energy dialog and reasoning toolbar (Layer Two - Quantitative)

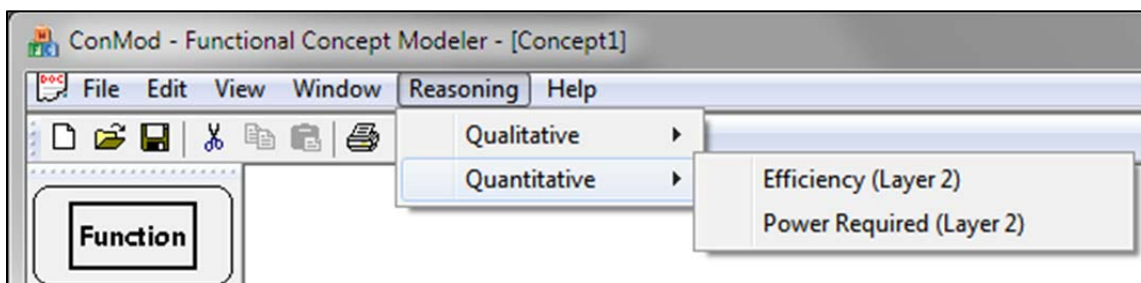


Figure 7.12: Reasoning menu options in ConMod (Layer Two - Quantitative)

The force and rate terms are generic placeholders for conjugate quantities, whose product defines power associated with many common energy forms. For example, the rate of mechanical work required to cause linear motion of a body (e.g., a slider in a guide) against a force (e.g., friction between the slider and a guide) is the product of the **force** applied and the **velocity** (rate of displacement) of the point of application of the force:  $P = F \times v$ . For rotational systems, e.g., in a rotating shaft, power transmitted is the product of the **torque** (analogous to force) and **angular speed** (rate of angular displacement):  $P = T \times \omega$ . The rate of electrical work done by current passing through a resistance is the product of the **voltage or electromotive force** (analogous to force) and the **current** (rate of charge):  $P = V \times I$  [132-134, 177]. Thus, in many forms of energy, specifically those that involve time rate of a quantity, the force term and the rate term can be used to compute power. It is recognized that this correspondence does not apply to many other energy forms. The purpose of using these conjugates instead of a more direct Power attribute in the CEnergy dialog box is to recognize that energy flows are often expressed in terms of indirect quantities that are more measurable and appropriate for specifying those energy types than using the power attribute directly. For example, engines are rated by the torque-speed characteristics and motors are specified by their operating voltage and RPM. However, the only conservable quantities in these devices are mass, energy, and momentum within isolated systems. Thus, in order to apply energy balance, the quantities used to specify the flows must be first used to compute power. The representation should ideally support specifying flows in terms of the physical quantities appropriate for specifying their form, yet it should be capable of computing

power from these quantities. The use of the force term and rate term is only an indication of that flexibility to be achieved in future extension of ConMod. The two fields in Figure 7.11a, when multiplied, specify the power quantity of the CEnergy instance.

To demonstrate irreversibility reasoning in the quantitative level, the Air Heater function structure of Figure 7.8 is reconstructed using the implementation of ConMod-2 with quantitative extension (hereafter referred to as ConMod-2q). This extension supports all modeling and reasoning supported by the previous layers. In addition, when quantitative reasoning is chosen from the menu by selecting Reasoning > Quantitative > Efficiency (Layer 2), the model displays the default values of power for each flow, appended to the right of the flow names with a default unit of watts (W). The force term and rate term fields in the Add Energy dialog (Figure 7.11a) are also enabled.

The model displaying the default power magnitudes of the Energy flows is shown in Figure 7.13. Each energy flow is assigned a power of 100 Watts, based on the default values of the force term (100 SI units) and the rate term (1 SI unit) assumed by the default dialog in Figure 7.11a. The only exception is Loss7, which is intentionally edited to have a power of negative 100 W (-100 W – highlighted with an ellipse in the figure), in order to illustrate the checks executed under quantitative reasoning.

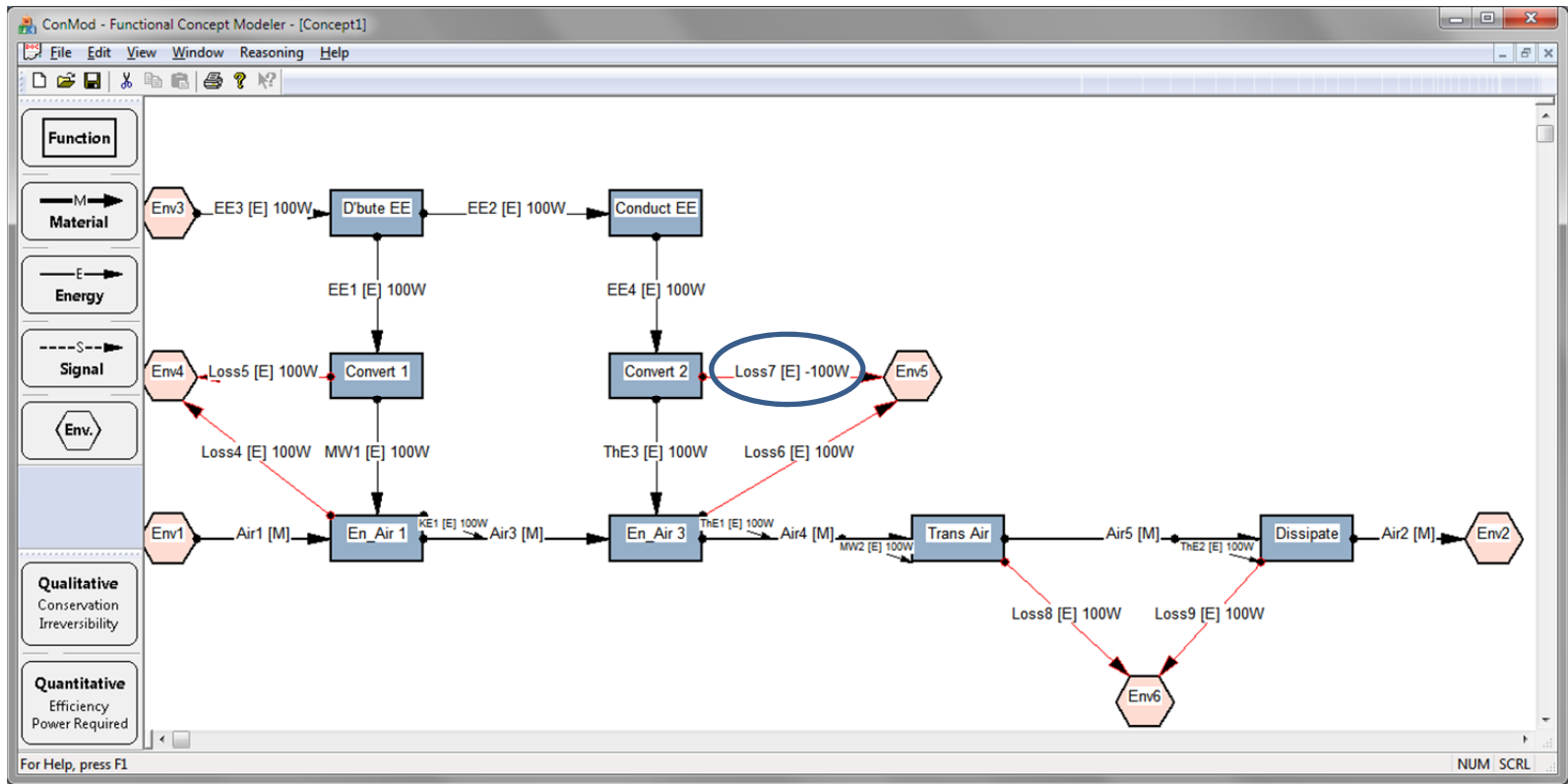
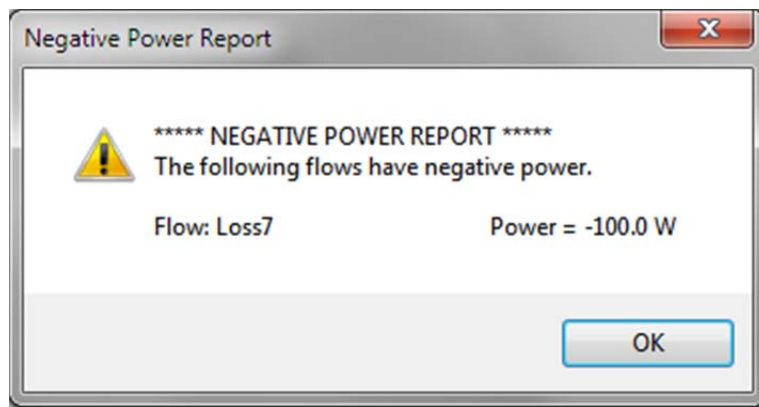
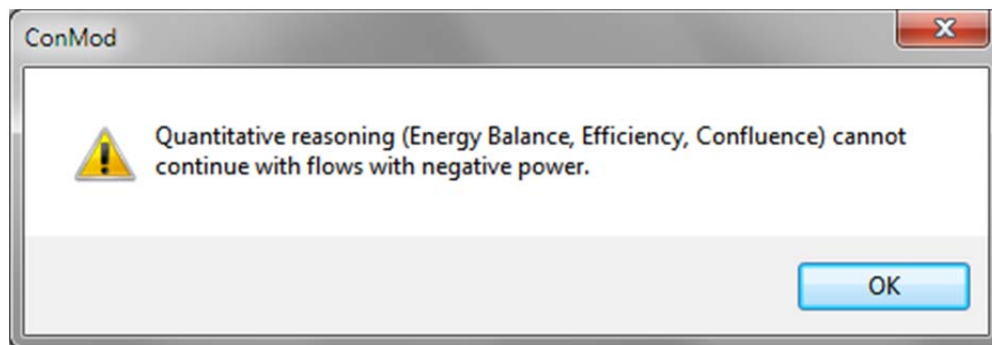


Figure 7.13: Quantitative model of the Air Heater using ConMod-2q, showing default power of Energy flows and one negative power value

Upon requesting quantitative reasoning, the algorithms are executed on the model in Figure 7.13 in three steps. Figure 7.14 shows the result from the first step. Loss7 is identified as a flow with negative power. The first step checks for existence of negative power values in energy flows. A negative power value can occur either by user editing (this case) or as a result of energy balance inference. At any time, if a flow with negative power is identified, the algorithm recognizes that the model is out-of-date. Further execution of reasoning (steps two and three) is aborted in this condition (Figure 7.15).



**Figure 7.14: Quantitative reasoning Step-1: Check for negative power magnitudes**



**Figure 7.15: Aborting reasoning steps under out-of-date model state**

The model is next edited to assign a power value of 100W to Loss7, thus eliminating flows with negative power. This corrected model state is shown in Figure 7.16. The corrected power value of 100 W is marked with an ellipse to highlight the difference with the previous model state.



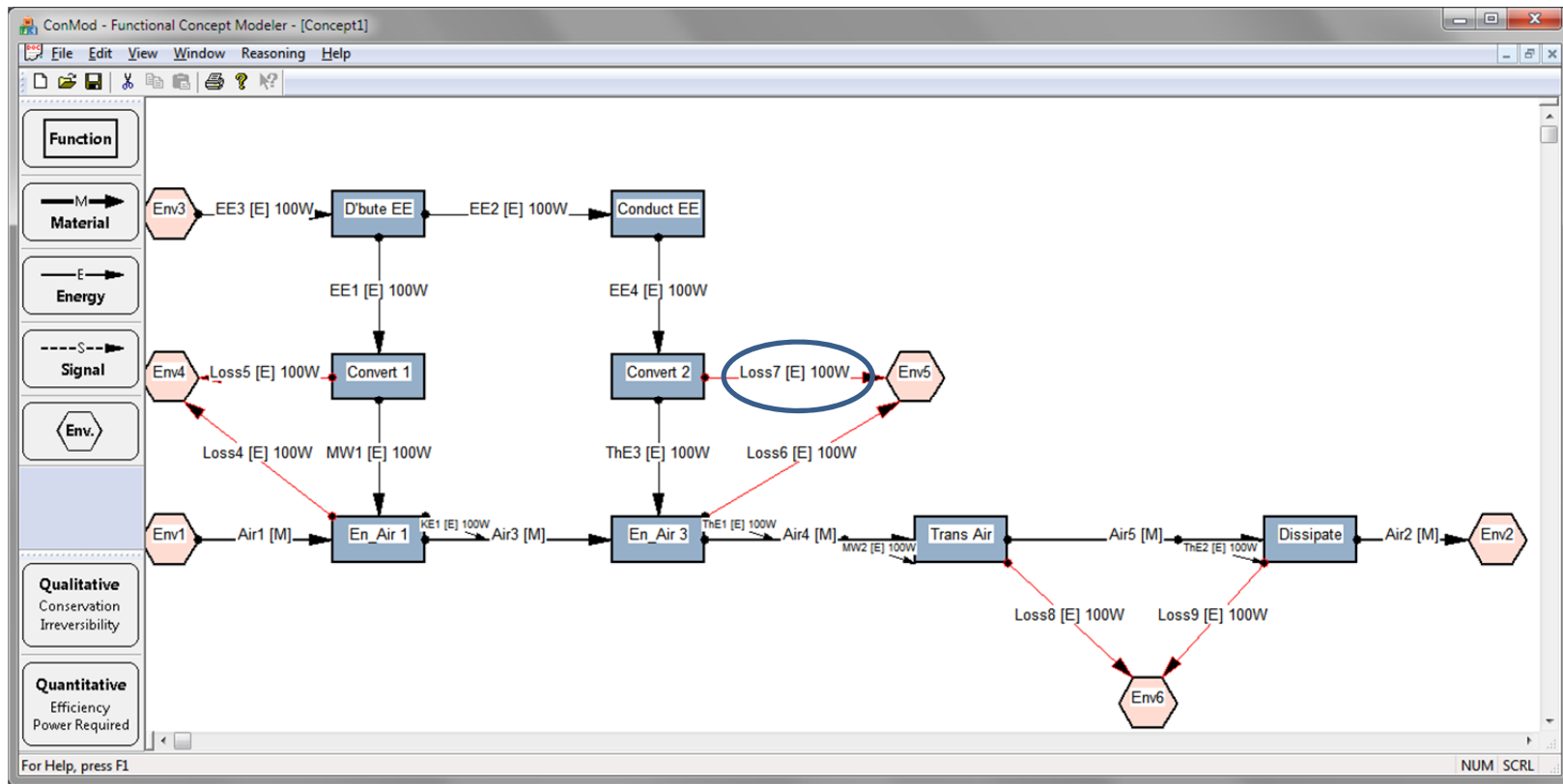
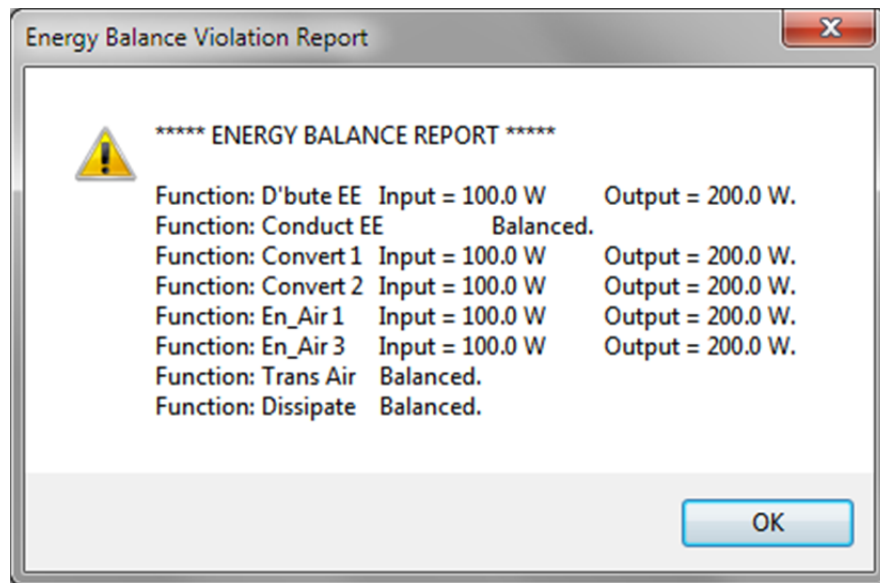


Figure 7.16: Quantitative model of the Air Heater using ConMod-2q, after correcting negative power values

Quantitative reasoning on this amended model produces the output shown in Figure 7.17 produced. The algorithm finds no flows with negative power and continues to the second step of quantitative energy balance. Since each energy flow has Power = 100 W, any function with unequal count of input and output energy flows is detected as an unbalanced function. The functions where a balance is found are also reported. An inspection reveals that the report in Figure 7.17 is correct to the model in Figure 7.16. Similar to the previous step, subsequent reasoning is aborted until quantitative energy balance is established in each function.



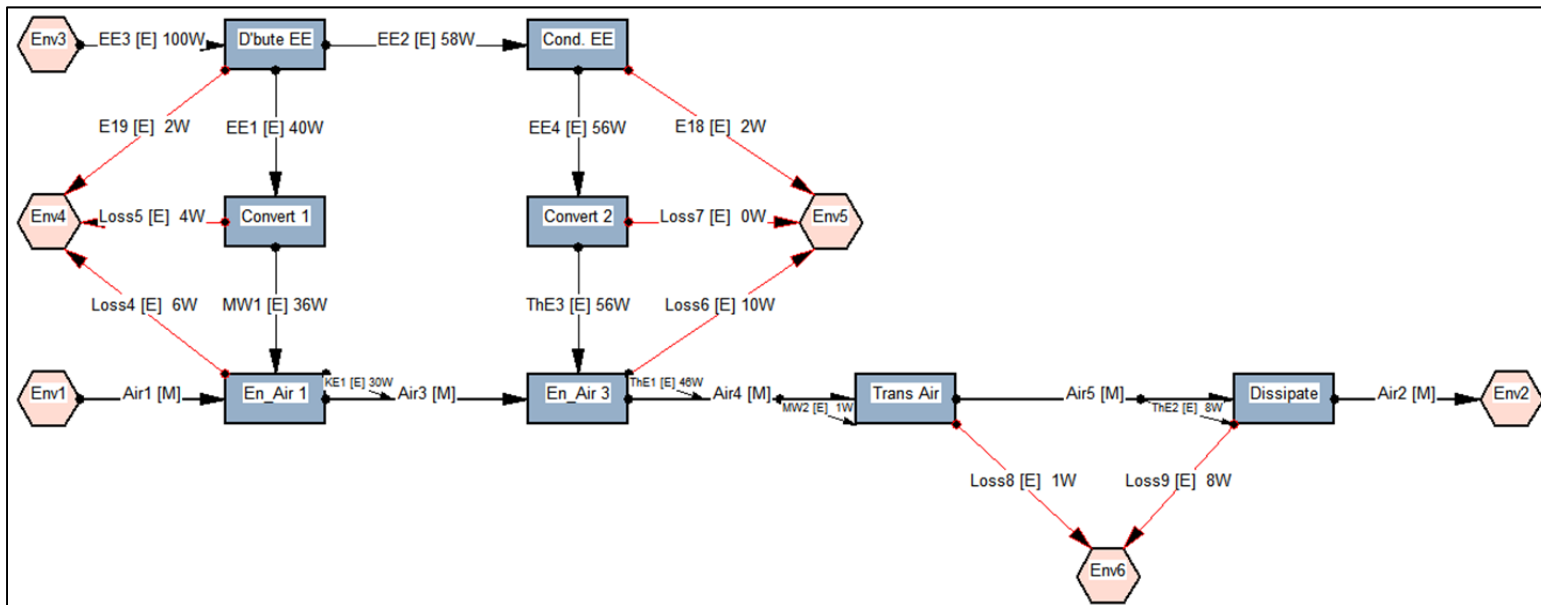
**Figure 7.17: Quantitative reasoning Step-2: Check for quantitative energy balance**

The model is next edited manually in order to achieve energy balance in each function. For ease of model review, the input power in EE3 is kept as 100 W. The resulting edited model is shown in Figure 7.18. The power magnitudes are chosen to

assign realistic values to the flows. For example, resistive loss due to the distribution and conduction of electrical energy are assigned only 2 watts each, while the En\_Air1 function (possibly executed by a fan) and the En\_Air3 function (possibly executed by a heater) have larger losses. The fan loses six watts out of the input 36 watts (~17% loss), while its fails to add to the air ten watts out of the 56 W of heat produced (~17% loss). The heater' functionality is shown in this model using two functions: Convert2 and En\_Air3. The first produces heat using electricity, while the second adds that heat to the air stream. The use of two functions for one device shows that the representation does not require maintaining a 1:1 mapping between functions and devices. In fact, in the early design stages, where this representation is intended to be used, the devices or embodiments of functions may not be known to the designer, and the representation should not require the designer to maintain such mapping.

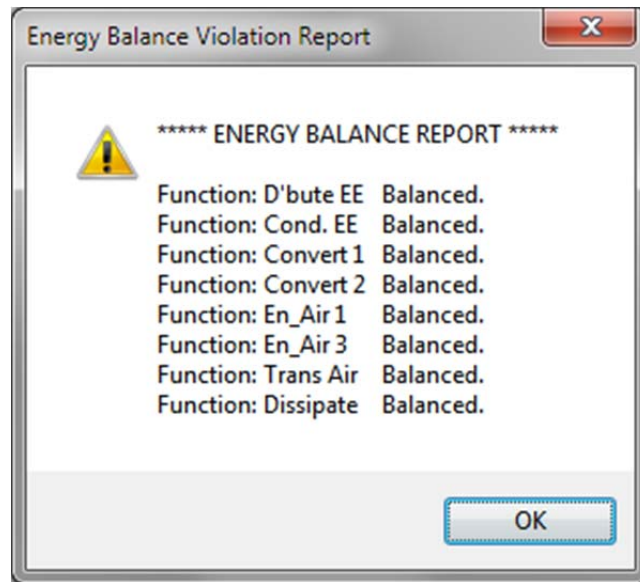
The loss from Convert2 is set to zero, since in resistive heating, all of the input electrical energy can be converted into heat (if the temperature is not high enough to trigger incandescence). This complete conversion of electrical energy to heat is not against the irreversibility principle. The principle requires losses in converting heat to work. When converting work to heat, all of the input work can be dissipated as heat. In resistive heating, the work is performed by the electric current (electrical work). This example shows another reason why it is useful to allow setting zero magnitudes to the loss flows. However, loss from En\_Air3 is shown as non-zero, as it is impossible for a heater to add heat to the flowing air stream without losing any part of it to colder

surroundings, since heat spontaneously flows from hotter to colder temperatures without requiring external work input.



**Figure 7.18: Air Heater model after manually ensured energy balance**

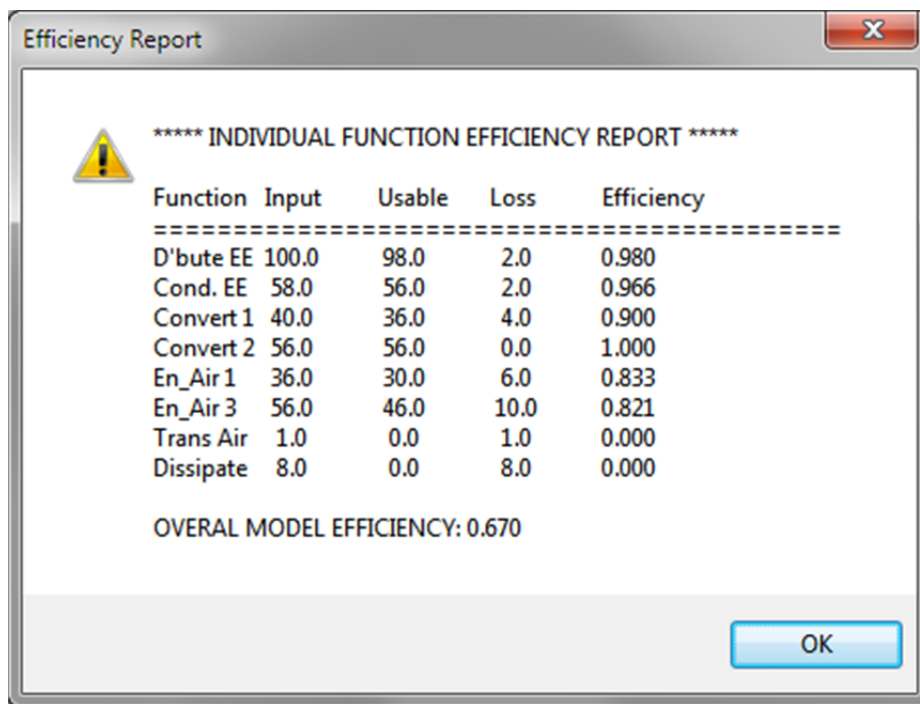
The result of the quantitative reasoning on the model of Figure 7.18 is shown next. In Figure 7.19, the reasoner determines that energy balance is achieved in the individual functions, and therefore, in the model as a whole. With this check passed, the reasoner continues to the third step of reasoning (Figure 7.20).



**Figure 7.19: Quantitative energy balance report: Step 2 with passing results**

Figure 7.20 shows the result of computing efficiency for the individual functions and the model as a whole. An inspection of the input, output, loss, and efficiency of the individual functions reveals that the reported numbers are correct based on the model. Individual efficiencies are computed as the ratio of power of all output energy flows that are not marked as residual to all input energy flows, residual or not. Further, the reasoner computes the efficiency of the model, as shown in the last line of Figure 7.20. The reported model efficiency, 67%, can be verified in two ways from the model. **First**, the total power of the output loss flows is  $(2 + 4 + 6 + 2 + 0 + 10 + 1 + 8)$  watts = 33 watts,

while the only input power is that of EE3, 100 watts. Thus, the efficiency is  $(100 - 33) / 100 = 67\%$ . **Second**, the total energy added to the air stream is in two steps, En\_Air1 adds 30 watts of kinetic energy, and En\_Air3 adds 46 watts of heat, totaling up to  $30 + 46 = 76$  watts. Out of this added energy, the air spends 1 watt to viscous resistance of the pipeline in the Transfer Air function and losses 8 watts of heat through the pipe walls in the Dissipate function, resulting in a net energy available in the outgoing Air2 flow as  $76 - (1 + 8) = 67$  watts. Thus, the efficiency is  $67/100 = 67\%$ . Both results agree with the reported model efficiency in Figure 7.20.



**Figure 7.20: Quantitative reasoning Step-3: Computing function-wise and model efficiency**

In summary, this chapter presents an extension to the first layer of the formal representation (Chapter 5) to support irreversibility-based reasoning at qualitative and quantitative levels. The extension of the representation involves the addition of three new attributes to existing classes. No change in the class structure is necessary. The enhanced reasoning ability due to this extension of the representation is demonstrated by implementing new algorithms in ConMod, which perform qualitative and quantitative model checking against the irreversibility principle and compute the function-wise and model-level efficiency. **Through these demonstrations, the representation's extendibility to support qualitative and quantitative reasoning on the irreversibility principle is validated.**

It should be mentioned that the three steps of quantitative reasoning—checking for negative power, checking for energy balance, and computing efficiency—are not the only quantitative reasoning that this representation can support at present. For example, the third step computes efficiency of functions from known values of input and output energy flows and identification of residues. Similarly, a ConMod model could be used to estimate the required power input to a model from the desired output power of output energy flows and the efficiencies of the functions. Similarly, if the available input power and the efficiency of the modeled functions were known, the model could be used to estimate the expected output power of the flows. Each of these reasoning types relies on the definition of efficiency as the ratio of total output usable power and the total input power. In each case mentioned, a different set of two variables are known and the third one is determined. Each of these reasoning actions can be supported by implementing



more algorithms in the ConMod software code, without any further extension of the representation. However, for illustration purposes, only the efficiency computing reasoning is implemented.

This ability to perform quantitative physics-based reasoning using function structures opens up the possibility of automating confluence-based reasoning in early design stages, although the need for a variational solving system is identified to support that reasoning. For example, in the computation of function-wise efficiency in ConMod-2q, the following parametric definition of efficiency is used:

$$\text{Efficiency} = (\text{Usable output power}) / (\text{Total input power})$$

This parametric relation can only be used when the terms in the right hand side are known and the efficiency (left hand side) is to be computed. However, when analyzing confluence, a designer may need to investigate the effect of changing the efficiency of a function on the usable output power, while the input remains constant. Solving this problem parametrically to determine the new output power would require the unknown variable (output) to be expressed in a new parametric form:

$$\text{Usable output power} = (\text{Total input power}) \times \text{Efficiency}$$

Similarly, if the input power is to be computed while efficiency is changed, another parametric expression for the same mathematical relation would be necessary. Instead of predicting the possible permutations of these parametric forms, a variational

solver can be used to solve these problems based on one declaration of the relation between the variables.

In the next chapter, the representation is further extended by proposing a set of function verbs. These verbs are defined using the existing representation and ensure physics-based concreteness of definitions. These verbs can be used as a vocabulary of elementary functional actions suitable for constructing function structures that can support more enhanced physics-based reasoning.

## CHAPTER 8. REPRESENTATION LAYER THREE: SEMANTIC LAYER: A PHYSICS-BASED VOCABULARY OF FUNCTION VERBS

The first two layers of the representation described in Chapter 5 through Chapter 7 rely on the six entity types: Function, Source, Sink, Material, Energy, and Signal, and the 33 grammar rules of Layer 1. Any function structure based on this representation must be defined with these vocabulary terms and grammar rules. While models so constructed are internally consistent, externally valid against conservation and irreversibility, and support formal reasoning, one limitation is that the models are entirely syntactic; they do not capture or support reasoning on the meanings of the model terms. For example, all reasoning would be equally applicable if the term Function was replaced with any other word, or if a function named Convert Energy did not convert anything. Constructing function models using **meaningful** terms not only increases the expressive power of the model to human interpreters, it makes modeling easier and more consistent, and when implemented in a formal representation, enables enhanced semantic reasoning, many of which are necessary to perform design tasks such as solution search, problem decomposition, model comparison, or similarity detection. This chapter begins to address this need by extending the representation by proposing a finite vocabulary of atomic actions (function verbs) to be used for function structure construction. These verbs are formally defined to capture **semantics** of those actions using their topology (types and count of flows attached) and additional grammar rules. This chapter verifies that this vocabulary provides adequate coverage over function modeling, by constructing models

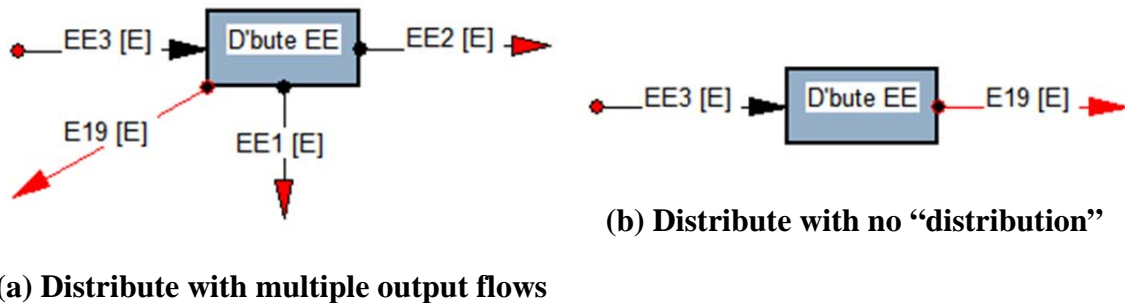
to describe fundamental processes of physics and the engineering sciences, and by modeling complete products in both descriptive and normative modes. However, demonstration of reasoning supported by the vocabulary is not presented in this chapter and is reserved for future efforts. In the following section, the need for capturing semantic information in models is further developed.

### **8.1 The Need for Semantic Information in Function Modeling and Reasoning**

The only entity to describe a transformative action in the current representation is Verb. The Verb class is shown to be able to describe many actions, such as distributing energy, converting energy, transferring energy, or energizing material flows (Figure 7.18). However, in each case, the function and its topological constructs—count and type of input and output flows—must be controlled manually to ensure that the modeled function and its attached flows truly describe the action the modeler intends. The grammar rules prevent constructs that violate entity definitions and the reasoner detects violations of natural laws, but these controls cannot ensure that a function’s topology describes—or carries the **meaning** of—a specific transformative action such as Distribute or Convert. It is therefore possible that the modeler adds a function with the intent to describe an action such as distribution or conversion and then edits the model to the point where the function does not describe the originally intended action. The reasoner cannot draw inferences on the **semantics** of the modeled terms to detect this type of errors.

For example, Figure 8.1a shows the Distribute EE function from Figure 7.18 and its attached flows as an isolated construct. This function is accepted by both the grammar

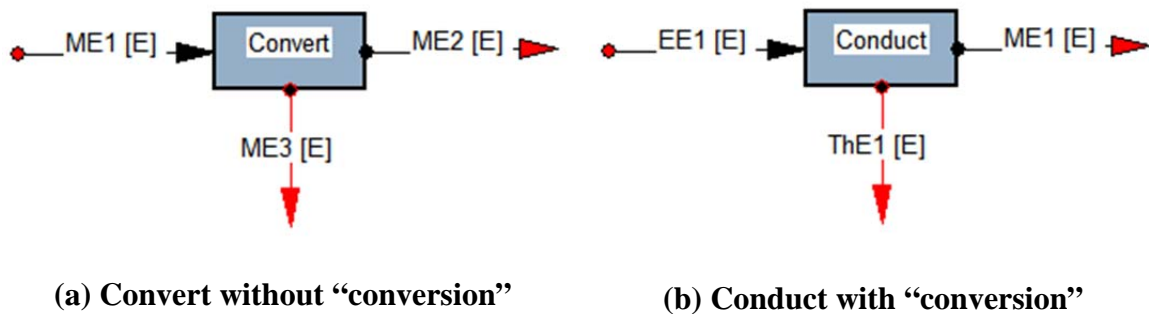
and reasoning algorithms of Layer 1. For this example, it is assumed from the name of the function that the original intent of adding this function was to describe a distribution of energy. The function is next edited (in isolation, as shown) by deleting its output flows, finally to arrive at the construct of Figure 8.1b. At no step, including Figure 8.1b, do the grammar rules or the reasoner detect this construct as a violation of modeling intent, despite that the model no longer describes any distribution: Figure 8.1b has only one outgoing flow.



**Figure 8.1: Lack of formalism to capture function semantics**

This behavior is not surprising, given that there is no data element available in the representation to store the original intent of distributing a flow. Notably, the GivenName attribute “D'bute EE” is only a string for identification, not used in any semantic reasoning. Similarly, Figure 8.2a shows an instance of Convert that does not convert anything: all of its output flows are of the same type (mechanical energy, ME) as the input. Instead, Figure 8.2b shows a function that changes the type of the incoming energy flow from electrical energy (EE) to mechanical (ME) and thermal (ThE), yet is not described as a Convert function. This behavior is also expected, since in addition to

the intent of the verb being omitted, the energy flow labels EE and ME are only strings for identification. The representation has no data element to capture that a flow marked as EE is in fact electrical energy and thus must have certain properties and behavior. In summary, the representation does not capture the semantics of functions and flows.



**Figure 8.2: Semantic inconsistency between function description and topology**

Allowing this type of errors not only makes the model wrong—as it does not describe what the modeler wants to describe—it allows the modeler to overlook violations of modeling intent, such as the missing output flows from Distribute shown above, which is more likely to happen in models of higher size and visual complexity. Consequently, the modeler can make incorrect inferences, such as underestimating the power requirement. The algorithms presented earlier cannot detect these errors, as these constructs do not violate conservation or irreversibility. It is anticipated that it would benefit the modeler to detect the violation of modeling semantics and support more enhanced reasoning about the concept based on this additional “knowledge” of semantics.

To detect these errors, the reasoner must compare the modeled constructs to internally stored **meanings** of the function verbs. To this end, the count and type of input and output flows attached to a function contains enough information to capture meaning. For example, in natural English, the verb distribute implies multiple flows being produced<sup>15</sup> (count), Convert implies the input and output flows being of different types<sup>16</sup> (type), and both Distribute and Conduct<sup>17</sup> imply no change of type within the scope of those verbs (type). Additionally, the definitions of these verbs in the Functional Basis vocabulary [26] also reveal similar semantic implications when objectively examined [48, 115, 116]. Thus, the count and type of input and output flows attached to a function are used here to surrogate the semantics of the function verb.

This approach of comparing model constructs with pre-stored semantic definitions is predicated upon a finite number of verbs to be captured in the representation. To this end, previous research indicates that a finite set of verbs can be used to describe a wide range of mechanical actions [26, 27, 90, 106]. Examples are the vocabulary of functions identified through engineering forensics of helicopters by Collins

---

<sup>15</sup> “to divide among several or many” [<http://www.merriam-webster.com/>, accessed on August 16, 2011]

<sup>16</sup> “to change from one form or function to another” [<http://www.merriam-webster.com/>, accessed on August 16, 2011]

<sup>17</sup> “to act as a medium for conveying or transmitting” [<http://www.merriam-webster.com/>, accessed on August 16, 2011]

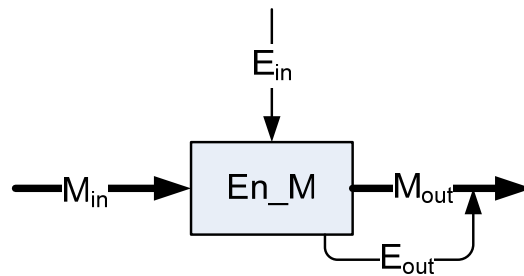
et. al. [89] and the Functional Basis vocabulary [26]. However, being identified through empirical observation (top-down approach), the verb definitions in these vocabularies have notional concreteness, rather than physics-based concreteness, as explained in Chapter 2. As a result, while these verbs are useful for human interpretation, they are not suitable for physics-based reasoning. It is anticipated in this research that even at the physics-based concreteness level, a finite set of verbs can be identified to describe mechanical devices and principles. **To this end, the aim of this chapter is to propose a new finite vocabulary of atomic function verbs that can describe mechanical functions with physics-based concreteness and to demonstrate that the vocabulary provides adequate modeling coverage over a variety of physics and engineering principles, phenomena, and devices.** Each verb proposed here is composed of one or more instances of functions and flows, and verb-specific grammar rules that capture the meaning of the verbs. This finite number of verbs and grammar rules does not imply that the total number of possible function structures is finite or that the representation has coverage over a finitely many concepts. Rather, it means that infinitely many models can be constructed using a finite number of individual verbs and a finite number of ways they can be arranged topologically. Before presenting this vocabulary, the prospects of using a finite vocabulary of formalized verbs in function structure modeling and reasoning are illustrated next with examples.

#### Potential Benefits of Using a Finite Set of Verbs in Function Modeling

A static function verb vocabulary can provide benefits in both modeling and reasoning. In modeling, a verb can be directly instantiated with its correct topological



construct, thus increasing consistency and speed of modeling. For example, an instance of the verb `Energize_M` used to model the addition of energy to a material (see Table 8.14) inputs one energy and one material flow, and produces one material and one energy flow so that the output energy is carried by the output material, as shown in Figure 8.3. This entire construct can be instantiated at once and thus can ensure correctness and consistency of this topology. Grammar rules can be written to prevent violating these constructs through erroneous editing.



**Figure 8.3: An instance of the proposed verb `Energize_M`**

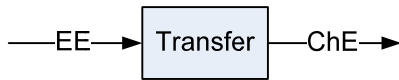
In terms of reasoning, the use of semantic information can improve accuracy of many automated design reasoning tasks, such as solution search, problem decomposition, model comparison and similarity detection, and computing model complexity. Solution search is discussed in design texts as a major activity potentially supported by function modeling [1, 2]. To automate this activity, a solution search system can be devised in two parts: (1) a database of solution principles, components, or subsystems, whose function structure graphs are also mapped to the devices and stored in a searchable format (similar to the \*.fst models produced in ConMod-2q), and (2) a search algorithm that can accept a function structure model in a similar format and search for solutions in the

database whose functional constructs (whole or part) matches with constructs in the input model. When a match is detected, the reasoner reasons that the stored device, whose function structure contains similar structure as the input model or its portions, can be a potential solution candidate and returns it.

Unless semantic information is formalized and implemented in the models, the search algorithm can at most use the topological similarity between the models to perform the search, without using information about the names or subtypes of the entities. For example, if the input construct is as shown in Figure 8.4a, the search may return the models in Figure 8.4b and Figure 8.4c from the database, since the only information usable for search is that the construct has one function with one input energy and one output energy. The facts that the function is a conversion of energy and the input and output types are ME and ThE are not usable, unless these words are formally defined in the representation. Additionally, it is impossible for the reasoner to detect that the Transfer function actually causes conversion and the Store function converts energy and supplies one of them, rather than storing something.



**(a) Input modeling construct**



**(b) Construct 1 stored in database**



**(c) Construct 2 stored in database**

**Figure 8.4: Two possible return values for solution search**

With semantic information captured in the vocabulary terms, these algorithms can have more information (data elements) to use in reasoning. For this solution search example, only those solutions that show the conversion from ME to ThE can be returned and thus, reasoning accuracy can be improved. It can be shown through similar exercises that vocabulary-based modeling can improve reasoning accuracy and efficiency for other activities such as similarity detection and model comparison. The proposed vocabulary of verbs is presented next.

## 8.2 Proposed Vocabulary of Atomic Function Verbs

The proposed vocabulary of atomic verbs contains three parts: (1) energy verbs, (2) material verbs, and (3) topologic verbs. Energy verbs are those that transform energy flows from one **state** (not type) to another, without using material flows. Material verbs are those that transform material flows and always involve energy flows, since the state of a material flow cannot be changed without exchanging energy, as per reasoning # 9 of

Table 4.15. Topologic verbs are logical operations, rather than mechanical actions, required to instantiate energy balance and mass balance declarations.

The selection of verbs in this taxonomy is based on an overall review of physical processes, especially the transport phenomena of heat and mass, with the intent of describing those processes as **actions** (functions) performed on material and energy flows. In outcome, it is recognized that only five basic actions are performed on energy:

1. Energy is converted from one form to another,
2. Energy is transferred from one location to another,
3. Energy is changed in quantitative specification,
4. Energy is stored in material media, and
5. Stored energy is released from material media.

The vocabulary of energy verbs is based on these actions, as discussed in Sections 8.2.1 and 8.2.2. Further, all actions on material flows are resolved into two basic actions:

6. Addition of energy to material, and
7. Removal of energy from material.

These two actions are the basis of the material verbs (Section 8.2.3). Topologic verbs, discussed in Section 8.2.4, are not mechanical actions, as mentioned earlier. The rationale behind reducing all processes into these seven actions rests on reviewing and modeling a large number of phenomena from various physics domains such as mechanics, hydrostatics, gravitation, elasticity, heat, acoustics, optics, electricity,

magnetism, radioactivity, and from the engineering sciences such as hydraulics, thermodynamics, and heat transfer. This accounting is omitted here for brevity. However, in order to support the choice of these actions as a basis for developing the vocabulary, the resulting vocabulary is tested for coverage over engineering processes and devices. The energy verbs at the primary level are presented next.

### **8.2.1 Energy Verbs – Primary Level**

The energy verbs are those that describe transformative actions on energy flows and are organized in a two-level taxonomy. At the primary level, five verbs are defined: (1) TypeChange\_E, (2) Transfer\_E, (3) Change\_E, (4) Store\_E, and (5) Supply\_E, based on the actions identified above. These verbs are described in Table 8.1 through Table 8.5. In each table, the first row introduces the verb's name, the second row provides a textual description of the verb for human interpretation of its purpose and action, the verb's semantics is captured in terms of the count and type of the attached flows in the third row, and the fourth row introduces the grammar rules (constraints) imposed on the flows as required by the semantics. Based on this information, the fifth row provides a formal definition of the verb using first order predicate logic statements in a set-based syntax. Finally, the sixth row identifies new flow attribute to be included in the representation in order to define these new verbs. Table 8.1 describes the verb TypeChange\_E.

**Table 8.1: Primary energy verb: TypeChange\_E**

<b>Verb name</b>	TypeChange_E		
<b>Textual definition</b>	To change the subtype of an energy flow		
<b>Input type and count</b>	One flow of a subtype of E (Energy)	<b>Output type and count</b>	One flow of a subtype of E (Energy)
<b>Grammar</b>	The input and output flows are necessarily of different subtypes.		
<b>Formal definition</b>	<pre> Class TypeChange_E : Verb {} // Inherited from class Verb { // Type of flows     List &lt;Energy*&gt; Input_E_List;    // Input energy list     List &lt;Energy*&gt; Output_E_List;  // Output energy list // Count of flows     Input_E_List = {E<sub>in</sub>};        // Only one input energy     Output_E_List = {E<sub>out</sub>};      // Only one output energy // Grammar constraints     E<sub>in</sub>.SubType ≠ E<sub>out</sub>.SubType; // The two flows are of different subtypes } </pre>		
<b>Attributes</b>	Energy subtype, such as electrical, mechanical, etc.		

TypeChange\_E is reserved for describing the conversion between energy forms, similar to the Functional Basis verb Convert<sup>18</sup> [26]. However, as defined in the Functional Basis, Convert involves more complex actions, such as the conversion of material and signal flows, the production of multiple flows, and residual output flows. As shown above, the definition of TypeChange\_E does not provide for residues or multiple output flows. Thus, the simultaneous production of heat and light by conversion from electrical energy in a light bulb cannot be modeled with one instance of TypeChange\_E. This behavior is intentional: TypeChange\_E is not intended to describe an entire conversion action. Rather, it is intended for describing only one fundamental action of changing one energy type into another. This character is true for all verbs presented in this chapter: they operate on the minimal number of flows and do not include residual energy. Additional grammar rules and algorithms are required to ensure that these atomic verbs can be combined to describe more complex concepts such as Convert and other notional verbs from the existing vocabularies, and to validate these models against the natural laws.

The definition of TypeChange\_E also indicates that a distinction between the subtypes of energy such as mechanical and electrical should be captured in the formal

---

<sup>18</sup> “To change from one form of a flow (material, energy, signal) to another. For completeness, any type of flow conversion is valid. In practice, conversions such as convert electricity to torque will be more common than convert solid to optical energy.”

representation. Developing a complete, consistent, and valid classification and formal definition of flows is not in the immediate focus of this research, as this research is more focused on formalizing functions at this stage. Thus, to provide for an energy classification, the types mentioned in the Functional Basis vocabulary are adopted with some modifications. Hydraulic and pneumatic energy are excluded, as these terms are **classes based on the carrier** of the energy, rather than form of energy. Hydraulic and pneumatic energy are means to transfer mechanical energy using a liquid or a gas carrier or medium. Human, biological, acoustic, and solar energy are dropped, as these terms are **classes based on the source** of energy rather than form. For example, biological energy obtained by burning wood contains multiple forms such as light and heat, and the energy stored in the wood can actually be better described as chemical energy that is released in combustion. Solar energy is a mix of different wavelengths of electromagnetic waves and not a distinct energy form than electromagnetic energy. Human energy can be a mix of many energy types. Acoustic energy is described as a form of mechanical energy conducted by a gaseous material. The optical and solar subgroups under electromagnetic energy are not included, since from a physics point of view, the only character distinguishing visible light from other electromagnetic waves is wavelength, and thus, the other waves in the electromagnetic spectrum such as radio waves, infrared, ultraviolet, x-rays, and gamma rays should be separately included. It is anticipated that scope of the representation will cover all properties of light used in the purview of general purpose mechanical engineering design that is covered under the general category electromagnetic energy. Radioactive and Nuclear energy are considered



as two different types. Nuclear is the type stored in an unstable nucleus, such as  $U^{238}$ . Radioactive energy is essentially electromagnetic energy that is released when an unstable nucleus undergoes decay and includes electromagnetic waves such as heat and gamma rays. Thus, the only terms from the Functional Basis considered in this table are mechanical, thermal, electrical, chemical, and electromagnetic. Each of these types is different energy forms and is governed by different physics. To continue with the vocabulary, Table 8.2 describes the verb Transfer\_E.

**Table 8.2: Primary energy verb: Transfer\_E**

<b>Verb name</b>	Transfer_E		
<b>Textual definition</b>	To change the location of an energy flow in geometric space		
<b>Input type and count</b>	One flow of a subtype of E (Energy)	<b>Output type and count</b>	One flow of a subtype of E (Energy)
<b>Grammar</b>	<p>The input and output flows are necessarily of the same subtype.</p> <p>The input and output flows have necessarily different locations.</p>		
<b>Formal definition</b>	<pre> Class Transfer_E : Verb {} // Inherited from class Verb { // Type of flows List &lt;Energy*&gt; Input_E_List; // Input energy list List &lt;Energy*&gt; Output_E_List; // Output energy list // Count of flows Input_E_List = {E<sub>in</sub>}; // Only one input energy Output_E_List = {E<sub>out</sub>}; // Only one output energy // Grammar constraints E<sub>in</sub>.SubType = E<sub>out</sub>.SubType; // No type change during transfer E<sub>in</sub>.Location ≠ E<sub>out</sub>.Location; // Location must change } </pre>		
<b>Attributes</b>	Energy flow location		

Table 8.3 describes the verb Change\_E.

**Table 8.3: Primary energy verb: Change\_E**

<b>Verb name</b>	Change_E		
<b>Textual definition</b>	To change the quantitative parameters of an energy flow without changing its type		
<b>Input type and count</b>	One flow of a subtype of E (Energy)	<b>Output type and count</b>	One flow of a subtype of E (Energy)
<b>Grammar</b>	The input and output flows are necessarily of the same subtype.  At least one parameter between the flows is necessarily of different value.		
<b>Formal definition</b>	<pre> Class Change_E : Verb {}           // Inherited from class Verb {   // Type of flows   List &lt;Energy*&gt; Input_E_List;     // Input energy list   List &lt;Energy*&gt; Output_E_List;    // Output energy list   // Count of flows   Input_E_List = {E<sub>in</sub>};         // Only one input energy   Output_E_List = {E<sub>out</sub>};       // No output energy   // Grammar constraints - NONE   E<sub>in</sub>.Subtype = E<sub>out</sub>.Subtype; // No change of type } </pre>		
<b>Attributes</b>	None		

The Change\_E function implies a change of quantitative specifications without a change of type. Since energy flows of different subtypes are specified with different parameter sets (e.g., torque and speed for rotational ME, current and voltage for EE), the formalization of Change\_E would require formalizing the parameters for each flow

subtype. At this stage, this representation does not formalize the flows classes or their parameters and thus it is not possible to formalize the definition of Change\_E for different energy subtypes beyond the generic definition above. Further, a change of location can be possible between the input and output energy flows, such as between the input ME and output ME flows of a gear box that are identified at the inlet and outlet of the box. Thus, at the present formalism, the definition of Change\_E looks similar to that of Transfer\_E, with the exception that location change is mandatory for Transfer\_E but not for Change\_E. Despite this similarity, it should be emphasized that Change\_E describes an entirely different mechanical action from the other verbs, although at the present formalism, it cannot fully accomplish that purpose. The definition above is a placeholder for future extension of Change\_E. Table 8.4 describes the next verb, Store\_E.

**Table 8.4: Primary energy verb: Store\_E**

<b>Verb name</b>	Store_E		
<b>Textual definition</b>	To store an energy flow in a material medium (part of system), where the medium behaves like a sink (singularity) and is able to receive an infinite amount of the energy flow type		
<b>Input type and count</b>	One flow of a subtype of E (Energy)	<b>Output type and count</b>	None
<b>Grammar</b>	None		
<b>Formal definition</b>	<pre> Class Store_E : Verb {}           // Inherited from class Verb { // Type of flows     List &lt;Energy*&gt; Input_E_List;   // Input energy list     List &lt;Energy*&gt; Output_E_List;  // Output energy list // Count of flows     Input_E_List = {E<sub>in</sub>};       // Only one input energy     Output_E_List = {∅};          // No output energy // Grammar constraints - NONE } </pre>		
<b>Attributes</b>	None		

Table 8.5 describes the verb Supply\_E. The secondary level of energy verbs are presented next.

**Table 8.5: Primary energy verb: Supply\_E**

<b>Verb name</b>	Supply_E		
<b>Textual definition</b>	To obtain energy from a material medium (part of system) , where the medium behaves like a source (singularity) and is able to release an infinite amount of the energy flow type		
<b>Input type, count</b>	None	<b>Output type, count</b>	One flow of a subtype of E (Energy)
<b>Grammar</b>	None		
<b>Formal definition</b>	<pre> Class Supply_E : Verb {}           // Inherited from class Verb { // Type of flows     List &lt;Energy*&gt; Input_E_List;    // Input energy list     List &lt;Energy*&gt; Output_E_List;   // Output energy list // Count of flows     Input_E_List = {∅};           // No input energy     Output_E_List = {E<sub>out</sub>};      // Only one output energy // Grammar constraints - NONE } </pre>		
<b>Attributes</b>	None		

### 8.2.2 Energy Verbs – Secondary Level

While the transfer of energy is captured in the primary verb Transfer\_E, in the study of heat transfer, three distinct mechanisms of transfer, governed by distinct principles and phenomena, are discussed: (1) conduction, (2) convection, and (3) radiation [128, 129]. A significant difference between these principles is in the interaction of energy with matter.

1. **Conduction** requires a material medium. While heat is transferred across the medium by means of the local motion (vibration) of the particles, the medium as a whole does not have the same net displacement as the conducted heat. Examples are the conduction of heat or electricity through a metal conductor, from a zone of higher temperature or electrical potential to a zone of lower temperature or electrical potential.
2. **Convection** also requires a material medium and the medium as whole moves by a net displacement to carry energy with it. Convection includes **diffusion** of the energy into the fluid from a boundary and transfer of the fluid in a process called **advection**. An example is the warming of water in a pot heated from the bottom, where heat is diffused from the bottom plate into the water, and reaches the top layers of water chiefly by the advection of hotter (and thus lighter) water molecules from the bottom to top.
3. **Radiation** does not require a medium at all, although properties of the intervening material or empty space, such as refractive index and transparency, influence the quantitative parameters of radiation such as speed of light and loss of intensity due to absorption. An example is the propagation of light through air, water, or empty space.

It is anticipated that by distinguishing between these three types of energy transfer, enhanced reasoning can be supported in early design. For example, qualitatively it can be inferred from an instance of radiation that no material medium is necessary in the embodiment of that function, while from an instance of convection, it can be inferred

that a fluid medium must be used, since only liquids and gasses can support advection. In future quantitative extensions, an instance of one of these types could be used to infer material properties pertaining to that type, such as conductivity for conduction and convection coefficient for convection.

Further, it is anticipated that the characteristic difference between these three mechanisms in terms of interaction of energy and matter can be conceptually extended to other forms of energy than heat, such as mechanical, electrical, or chemical. With this extension, conduction, convection, and radiation can be used as general subtypes of energy transfer. This anticipation is verified next by attempting to describe transfer processes of various energy forms analogous to conduction, convection, and radiation.

#### Verification of Three Types of Transfer for Different Energy Types

In Table 8.6, five actions—Conduct, Diffuse, Advect, Radiate, and Store—are five columns on the right. Each cell contains one or more examples of the action mentioned in the column header on a given type of energy (column 1) and using a given type of material (column 2). For example, the top left cell describes the action Conduct for Mechanical Energy (ME), using material of the type solid (S). The first four actions are the potential subtypes of Transfer\_E under examination. Store is included to discover if all energy types can be stored, so that grammar rules for the verb Store\_E can be written accordingly. Supply is not separately shown. For each type stored, an opposite phenomenon can be used to describe supply of the energy. For the material types (column



2), the three basic **phases** of material—solid (S), liquid (L), and gaseous (G)—are considered.

**Table 8.6: Three modes of transfer and storage for different energy types**

E	M	Conduct	Diffuse	Advect	Radiate	Store
Mechanical Energy (ME)	Solid	A <b>rotating shaft</b> conducts <b>mechanical work</b> from one location (end) to another location (end). The shaft rotates locally, but does not undergo net displacement between the ends as the energy.	Kinetic energy is handed over (diffused) from a moving <b>billiard ball</b> to a static ball in collision and across multiple balls in a <b>Newton pendulum</b> .	A <b>projectile</b> (e.g., a bullet) carries <b>kinetic energy</b> from one location to another, and must have displacement between the locations to transfer energy.	X Mechanical energy cannot be radiated, as it cannot be transferred without using material.	<b>Gravitational PE</b> is stored in a raised solid object, such as a <b>raised hammer</b> . <b>Elastic Strain E</b> is stored in a solid object (e.g., <b>spring</b> ) by elastically deforming its geometry.
	Liquid	<b>Water</b> in a pipeline conducts mechanical energy during <b>surging</b> or <b>water hammer</b> . An open water surface conducts ME as <b>waves</b> . In both cases, a wave propagates without net displacement of water.	Kinetic energy is diffused from a water jet to the buckets of a <b>Pelton wheel</b> during collision.	<b>Water</b> exiting a nozzle and hitting the buckets of a <b>Pelton wheel</b> advects kinetic energy, as the water itself must move to cause the transfer from the nozzle to the bucket.		A <b>soap bubble</b> stores <b>surface energy</b> (potential energy). Water stored behind a <b>dam</b> has stored <b>gravitational potential energy</b> .
	Gaseous	Air conducts ME as <b>sound waves</b> and <b>shockwaves</b> . Both are perceived by humans as sound, when the frequency is in the audible range (20-20,000 Hz).	Wind hitting the blades of a <b>windmill</b> diffuses its kinetic energy to the blades.	Air hitting the blades of a <b>windmill</b> advects kinetic energy. Both have net displacement to carry energy to the blades.		<b>Mechanical work</b> is stored in a tank of <b>compressed air</b> when the air is compressed. During expansion, this stored work is released as exergy.

E	M	Conduct	Diffuse	Advect	Radiate	Store
Thermal Energy (ThE)	Solid	Heat is conducted through <b>heat exchanger plates</b> and <b>radiator fins</b> . The plates and fins do not move, while the heat is transferred.	Heat is diffused from a hot solid to a cold solid in thermal contact, such as from the tip of a <b>soldering iron</b> to solid solder.	Solids cannot support advection by molecule motion, but the whole solid can move and carry heat. <b>Shotgun pellets</b> are made by dropping molten lead drops from a height into water to cool and freeze. The heat rejected to water is advected by the pellets across the height.	Radiated heat is classified as electromagnetic energy (EME).	Heat is stored in all solid objects as internal energy, indicated by its temperature. It is better received when it is extracted from the solid as <b>sensible heat</b> , which causes measurable drop of temperature.
	Liquid	When a <b>liquid column is heated at the top layers</b> , the only means for heat to transfer to the bottom layer is by conduction. Due to a negative density gradient (hotter liquid at the top), convection cannot ensue.	Heat is diffused from hot liquid to solid in thermal contact, such as from hot coolant to the walls of a <b>radiator</b> in a car.	When a <b>liquid column is heated at the bottom</b> , heat is transferred to the top through advection, which starts once the temperature difference and height of the column is large enough for the column to collapse.		Heat is stored in all liquids as <b>internal energy</b> , indicated by its temperature. It can be removed as <b>sensible</b> or <b>latent heat</b> , depending on whether the liquid is above or at its freezing point.
	Gaseous	When a <b>gas column is heated at the top layers</b> , the only means for heat to transfer to the bottom layer is by conduction.	Heat is diffused from hot gas to a solid in thermal contact, such as from the hot exhaust gas to the walls of the <b>exhaust pipe</b> in a car.	When a gas column (e.g., the <b>air column inside a chimney stack</b> ) is heated at the bottom, the main transfer process is convection, similar to liquids. The gas particles themselves move to the top to carry heat with them.		Heat is stored in vapors as <b>internal energy</b> , as indicated by its temperature. It can be removed as <b>sensible</b> or <b>latent heat</b> , depending on whether the vapor is above or at its condensation point.

E	M	Conduct	Diffuse	Advect	Radiate	Store
Electrical Energy (EE)	Solid	Electrical energy is conducted through <b>wires</b> . The conductor does not need to move to transfer the energy.	Electrical energy is diffused from one conductor to another in electrical contact, such as between the contacts of a <b>switch</b> .	EE is convected by <b>solid toner particles</b> in laser printers from the drum to the paper. The particles are charged and are transferred due to attraction of the oppositely charged paper. A small current is established by the flow of the particles. The energy of this current is advected by the displacement of the toner particles.		Static electricity is stored in solid objects such as toner particles, charged capacitor plates, and the leaves of an electroscope. The storage of electric energy in batteries is not mentioned, since the stored form is chemical energy.
	Liquid	Electrical energy is conducted through liquids such as <b>mercury</b> . The liquid does not move in order to conduct the energy.	Electric energy is diffused between a liquid and a solid in electrical contact, such as between the electrolyte and an electrode in <b>electroplating</b> .	<b>Charged paint droplets</b> in spray painting advect electrical energy from the nozzle to the work surface by physically moving in space.	Electrical energy is transferred through empty space or air by <b>induction</b> , such as between the high-voltage and low-voltage coils of a <b>transformer</b> .	Static electricity is stored in liquids, such as in <b>gasoline</b> during transport due to friction against the inside of a tank and in <b>water droplets in clouds</b> .
	Gaseous	Electric energy is conducted (discharged) through gas in <b>cathode ray discharge</b> tubes and through air during <b>lightning</b> . Although flow of charged gas particles is involved, the gas does not undergo net displacement.	Charged gas in <b>plasma arc welding</b> is discharged in contact with a solid.	Gasses in plasma state (e.g., in <b>plasma arc welding</b> ) convect electrical energy, where gas particles have to move in space to transfer the energy.	A load connected across the low voltage side consumes EE transferred from the high-voltage side without using a conductor.	Static electric charge stored in gas particles.

E	M	Conduct	Diffuse	Advect	Radiate	Store
Chem. Energy (ChE)	Solid	<p style="text-align: center;">X</p> <p>Chemical energy cannot be conducted through material media. ChE is energy locked in the chemical bonds of the material and thus always needs a material carrier. It cannot be transferred without moving the material that stores it.</p>	<p style="text-align: center;">X</p> <p>Chemical energy cannot be diffused</p>	ChE is convected by the flow of <b>pulverized coal</b> in a furnace in a boiler plant. The coal must to transfer its stored ChE.	<p style="text-align: center;">X</p> <p>Chemical energy cannot be radiated.</p>	Chemical energy is stored in the molecular bonds of <b>carbon</b> and in the <b>solid electrolytes of a rechargeable cell</b> during charging.
	Liquid			ChE is convected by the flow of <b>gasoline</b> in an engine.		Chemical energy is stored in the atomic bonds of liquid hydrocarbons, such as <b>liquefied propane</b> .
	Gaseous			ChE is convected by the flow of <b>propane</b> gas in a grill.		Chemical energy is stored in the atomic bonds of gaseous hydrocarbons, such as <b>gaseous propane</b> .

E	M	Conduct	Diffuse	Advect	Radiate	Store
Electromagnetic Energy (EME)	Solid	<p><b>Conduct through solids:</b> The only physical process of transferring EME is radiation. However, the process of transmitting light through an <b>optical fibers</b> and cables can be modeled as conduction, since the cable does not undergo any displacement.</p>	<p><b>Diffuse: X</b> The entrance of light from one medium to another can be viewed as diffusion, but since the medium does not play any role in transferring the light, this view is considered inappropriate in function modeling.</p>	<p><b>Advect: X</b> The only means of transferring EME is radiation.</p>	<p><b>Radiate:</b> EME is transferred as radiation in the propagation of radio waves, infrared light (ThE), visible light (optical energy, OpE), X-rays, and gamma rays.</p>	<p><b>Store: X</b> Electromagnetic energy cannot be stored in any medium <b>in electromagnetic form</b>. It can be converted and stored in other forms, as in <b>photovoltaic cells</b> or <b>solar panels</b> that convert EME from the sun into EE.</p>
	Liquid					
	Gaseous					

As shown in Table 8.6, the characteristics feature of the three energy transfer mechanisms in terms of interaction of energy with material can be extended to other energy form than heat using the concepts of conduction (medium does not move), diffusion (transfer across media in contact), advection (medium carries the energy), and radiation (medium not required) studied in heat transfer. However, as seen in the material verbs, diffusion can be modeled as the energizing of material (e.g., adding heat to colder fluid by a hotter plate) or de-energizing of material (e.g., removing heat from a hotter fluid to a colder plate), and advection can be modeled using the carrier flow relation, as energy carried by material. Thus, the two phenomena within convection, diffusion and advection, do not need to be captured as separate verbs. Thus, the only secondary verbs required under Transfer\_E are (1) Conduct\_E and (2) Radiate\_E, which are described next. Table 8.7 describes the verb Conduct\_E.

**Table 8.7: Primary energy verb: Conduct\_E**

<b>Verb name</b>	Conduct_E		
<b>Textual definition</b>	To transfer energy using a material medium, while the medium does not undergo a net displacement between the locations of energy transfer		
<b>Input type, count</b>	One flow of a subtype of E	<b>Output type, count</b>	One flow of a subtype of E
<b>Grammar</b>	All inherited grammar from Transfer_E  The input energy subtype cannot be Chemical Energy, as per Table 8.6		
<b>Formal definition</b>	<pre>Class Conduct_E : Transfer_E {}           // Inherited from class Transfer_E { // Grammar constraints     E<sub>in</sub>.SubType ≠ ChE;           // Chemical energy cannot be conducted }</pre>		
<b>Attributes</b>	None		

Table 8.8 describes the verb Radiate\_E.



**Table 8.8: Primary energy verb: Radiate\_E**

<b>Verb name</b>	Radiate_E		
<b>Textual definition</b>	To transfer energy without using a material medium		
<b>Input type, count</b>	One flow of a subtype of E	<b>Output type, count</b>	One flow of a subtype of E
<b>Grammar</b>	<p>All inherited grammar from Transfer_E</p> <p>The input energy subtype can be only electrical, thermal, or electromagnetic, as per Table 8.6</p>		
<b>Formal definition</b>	<pre>Class Radiate_E : Transfer_E {} // Inherited from class Transfer_E { // Grammar constraints     E<sub>in</sub>.SubType ∈ (ThE ∪ EE ∪ EME); // Only ThE, EE, and EME can be radiated }</pre>		
<b>Attributes</b>	None		

As seen above, the only difference between the primary verb Transfer\_E and its children is in the additional grammar rules that controls which subtypes of energy can be conducted or radiated. The identification of these transfer process for various energy types identified in Table 8.6 is used to derive these grammar rules. In Conduct\_E, the conducting material is not included as an input flow, since by definition of function (Section 1.2.7) and flow (Section 1.2.6), a function is an action **performed by the device**, and a flow is an entity **that the device acts upon** [178]. Since Conduct\_E is a function executed by the modeled system, the medium of conduction must be a

component of the system, and therefore should not be included as a flow. To this end, previous research [179] indicates that reasoning such as those involved in similarity detection between function models is inaccurate when the distinction between system and flows is ignored. Table 8.9 summarizes the energy verbs in a taxonomy, and includes their textual definitions tables where they are formally defined. The next subsection describes the material verbs.

**Table 8.9: Summary of energy verbs and their description tables**

<b>Primary</b>	<b>Secondary</b>	<b>Textual Description</b>	<b>Defined in Table</b>
TypeChange_E		To change the subtype of an energy flow	Table 8.1
Transfer_E		To change the location of an energy flow in geometric space	Table 8.2
	Conduct_E	To cause a change of location of an energy instance using a medium, where the medium does not change location	Table 8.7
	Radiate_E	To cause a change of location of an energy instance without using a medium or a carrier, although physical properties of the medium intervening the two locations may influence the process	Table 8.8
Change_E		To change the quantitative parameters of	Table 8.3

	an energy flow without changing its type	
Store_E	To store an energy flow in a material medium (part of system), where the medium behaves like a sink (singularity) and is able to receive an infinite amount of the energy flow type	Table 8.4
Supply_E	To obtain energy from a material medium (part of system) , where the medium behaves like a source (singularity) and is able to release an infinite amount of the energy flow type	Table 8.5

### 8.2.3 Material Verbs

Material verbs are those that perform transformative actions on material flows. As discussed earlier in this section, two basic actions involving materials are identified from the review of physical processes, which are translated into verbs next. Table 8.10 describes the verb Energize\_M.

**Table 8.10: Material verb: Energize\_M**

<b>Verb name</b>	Energize_M		
<b>Textual definition</b>	To add energy to a material flow		
<b>Input type, count</b>	One flow of a subtype of E One flow of type M	<b>Output type, count</b>	One flow of a subtype of E One flow of type M
<b>Grammar</b>	<p>The input energy flow must not be a carried flow.</p> <p>The output energy flow must be carried by the output material flow.</p> <p>The subtype of the input energy flow must be the same as the output energy.</p> <p>No restriction on the subtypes of the material flow, since phase change between solid, liquid, or vapor may occur as result of energizing.</p>		

<b>Formal definition</b>	<pre> Class Energize_M : Verb {} // Inherited from class Verb { // Type of flows List &lt;Energy*&gt; Input_E_List; // Input energy list List &lt;Energy*&gt; Output_E_List; // Output energy list List &lt;Material*&gt; Input_M_List; // Input material list List &lt;Material*&gt; Output_M_List; // Output material list  // Count of flows Input_E_List = {E<sub>in</sub>}; // Exactly one input energy Output_E_List = {E<sub>out</sub>}; // Exactly one output energy Input_M_List = {M<sub>in</sub>}; // Exactly one input material Output_M_List = {M<sub>out</sub>}; // Exactly one output material  // Grammar constraints E<sub>in</sub>.SubType = E<sub>out</sub>.SubType; // No type change of energy E<sub>in</sub>.Carrier = ∅; // Input energy must not be a carried flow E<sub>out</sub>.Carrier = M<sub>out</sub>; // Energy must be added to material } </pre>
<b>Attributes</b>	None

Table 8.11 describes the material verb DeEnergize\_M.

**Table 8.11: Material verb: DeEnergize\_M**

<b>Verb name</b>	DeEnergize_M		
<b>Textual definition</b>	To remove energy from a material flow		
<b>Input type, count</b>	One flow of a subtype of E One flow of type M	<b>Output type, count</b>	One flow of a subtype of E One flow of type M
<b>Grammar</b>	<p>The input energy flow must be carried by the input material flow.</p> <p>The output energy must not be a carried flow.</p> <p>The subtype of the input energy flow must be the same as the output energy.</p> <p>No restriction on the subtypes of the material flow, since phase change between solid, liquid, or vapor may occur as result of energizing.</p>		

<b>Formal definition</b>	<pre> Class DeEnergize_M : Verb {} // Inherited from class Verb { // Type of flows List &lt;Energy*&gt; Input_E_List; // Input energy list List &lt;Energy*&gt; Output_E_List; // Output energy list List &lt;Material*&gt; Input_M_List; // Input material list List &lt;Material*&gt; Output_M_List; // Output material list  // Count of flows Input_E_List = {E<sub>in</sub>}; // Exactly one input energy Output_E_List = {E<sub>out</sub>}; // Exactly one output energy Input_M_List = {M<sub>in</sub>}; // Exactly one input material Output_M_List = {M<sub>out</sub>}; // Exactly one output material  // Grammar constraints E<sub>in</sub>.SubType = E<sub>out</sub>.SubType; // No type change of energy E<sub>in</sub>.Carrier = M<sub>in</sub>; // Energy must be removed from material E<sub>out</sub>.Carrier = ∅; // Ouput energy must not be carried } </pre>
<b>Attributes</b>	None

Similar to the energy verbs, the material verbs also input and output the minimal number of flows that describe the action intended and does not provide for residual flows. As mentioned earlier, reasoning algorithms must be written in addition to these definitions to ensure model validity against the principles of conservation and irreversibility. In the next subsection, the topologic verbs are described.

### 8.2.4 Topologic Verbs

Two topologic verbs—logical branch and logical unite—are described next. These verbs are not mechanical actions. Rather, these verbs are necessary to instantiate user-defined declarations of energy balance and mass balance between energy and mass

flows in the model. As shown in the next section, when used in conjunction with the other verbs presented, these verbs allow modeling residues and the branching of flows.

Table 8.12 describes the verb Logical\_Branch.

**Table 8.12: Topologic verb: Logical\_Branch**

<b>Verb name</b>	Logical_Branch		
<b>Textual definition</b>	To state that the mass or energy parameter of one flow (input) equals the sum of that parameter of a set of multiple other flows (output)		
<b>Input type, count</b>	Exactly one flow, E or M	<b>Output type, count</b>	At least one flow, E or M
<b>Grammar</b>	<p>The input flow is either a material or an energy, but not both</p> <p>All output flows are of the same subtype as the input</p> <p>For material, the input mass equals the sum of the output masses</p> <p>For energy, the input power equals the sum of the output powers</p>		



<b>Formal definition</b>	<pre> Class Logical_Branch : Verb {} // Inherited from class Verb { // Type of flows List &lt;Noun*&gt; Input_List; // Input list List &lt;Noun*&gt; Output_List; // Output list // Count of flows Input_List = {I<sub>1</sub>}; // Exactly one input flow Output_List = {O<sub>1</sub>, O<sub>2</sub>, O<sub>3</sub>...O<sub>n</sub>}; // Multiple output flow allowed n ≥ 1; // At least one output required // Grammar constraints I<sub>1</sub>.SubType = (M ∪ E); // Only material and energy can be balanced ∀i, O<sub>i</sub>.SubType = I<sub>1</sub>.SubType; // No change of subtypes allowed (I<sub>1</sub>.SubType = M) → ( I<sub>1</sub>.mass = ∑<sub>i=1</sub><sup>n</sup> O<sub>i</sub>.mass ); // Mass balance (I<sub>1</sub>.SubType = E) → ( I<sub>1</sub>.power = ∑<sub>i=1</sub><sup>n</sup> O<sub>i</sub>.power ); // Energy balance } </pre>
<b>Attributes</b>	None

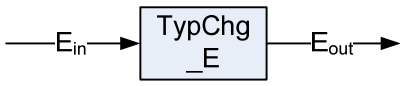
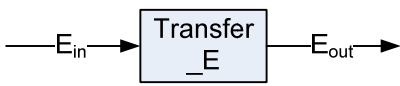
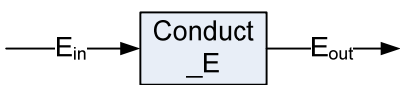
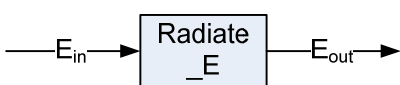
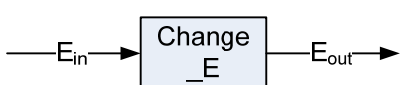
As seen from the above definition, the type of the flow is not changed. In addition, no transformative action is modeled using this verb. The purpose is to provide a means to declare that the mass or energy of one flow is conserved as the mass or energy of several other flows. Use of this verb is shown in the next section. Table 8.13 describes the verb Logical\_Unite.

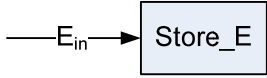
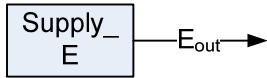
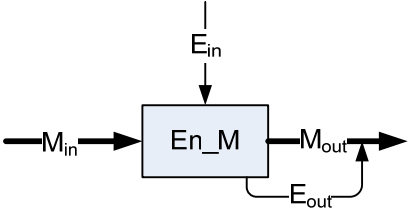
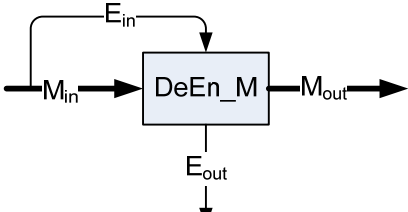
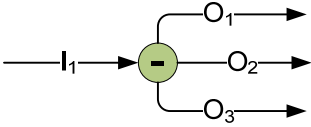
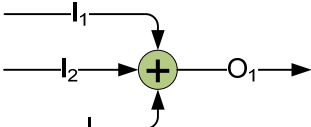
**Table 8.13: Topologic verb: Logical\_Unite**

<b>Verb name</b>	Logical_Unite		
<b>Textual definition</b>	To state that the mass or energy parameter of a set of multiple flows (input) equals the sum of that parameter of another flow (output)		
<b>Input type, count</b>	At least one flow, E or M	<b>Output type, count</b>	Exactly one flow, E or M
<b>Grammar</b>	<p>The output flow is either a material or an energy, but not both</p> <p>All input flows are of the same subtype as the output</p> <p>For material, the sum of the input masses equals the output mass</p> <p>For energy, the sum of the input powers equals the output power</p>		
<b>Formal definition</b>	<pre> Class Logical_Unite : Verb {} // Inherited from class Verb { // Type of flows List &lt;Noun*&gt; Input_List; // Input list List &lt;Noun*&gt; Output_List; // Output list // Count of flows Input_List = {I<sub>1</sub>, I<sub>2</sub>, I<sub>3</sub>...I<sub>n</sub>}; // Multiple input flow allowed Output_List = {O<sub>1</sub>}; // Exactly one output flow n ≥ 1; // At least one input required // Grammar constraints O<sub>1</sub>.SubType = (M ∪ E); // Only material and energy can be balanced ∀i, I<sub>i</sub>.SubType = O<sub>1</sub>.SubType; // No change of subtypes allowed (O<sub>1</sub>.SubType = M) → (∑<sub>i=1</sub><sup>n</sup> I<sub>i</sub>.mass = O<sub>1</sub>.mass); // Mass balance (O<sub>1</sub>.SubType = E) → (∑<sub>i=1</sub><sup>n</sup> I<sub>i</sub>.power = O<sub>1</sub>.power); // Energy balance } </pre>		
<b>Attributes</b>	None		

Before applying the verbs presented in this section to test their coverage over physical principles and devices, the verbs are summarized, along with their representative graphical symbols, in Table 8.14. In each verb symbol, the flow names are written to match the name declared in the formal definition, for ease of reference. This table also shows all the verbs and their taxonomy together and provides pointers to their definition tables. In the following section, this vocabulary is applied to model physical principles and devices to test the coverage of this vocabulary.

**Table 8.14: The proposed physics-based verbs and their graphical symbols**

Part	Primary Verb	Secondary Verb	Ref. Table	Graphical Symbol
<b>Energy verb</b>	TypeChange_E		Table 8.1	
	Transfer_E		Table 8.2	
		Conduct_E	Table 8.7	
		Radiate_E	Table 8.8	
	Change_E		Table 8.3	

Part	Primary Verb	Secondary Verb	Ref. Table	Graphical Symbol
	Store_E		Table 8.4	
	Supply_E		Table 8.5	
Material verb	Energize_M		Table 8.10	
	DeEnergize_M		Table 8.11	
Topologic verb	Logical_Branch		Table 8.12	
	Logical_Unite		Table 8.13	

The Logical\_Branch and Logical\_Unite verb instances are together referred to as balance nodes of function structure models presented hereafter. As seen from the symbols, both Store\_E and Supply\_E functions violate derivational conservation, as they do not maintain energy balance between the input and output sides. While this violation may lead to the identification of those flows as orphan or barren by the reasoning algorithms of ConMod-2q, this modeling construct is consistent with physics as energy can be stored in material bodies and later released. For example, in a rechargeable battery, chemical energy is stored during charging and liberated during discharging. Thus, derivational reasoning for this semantic layer should be waived for the special cases of Store\_E and Supply\_E.

In summary, this chapter presents a finite vocabulary of verbs that are claimed to be atomic actions performed by mechanical devices and can be used in modeling engineering devices and phenomena as function structure models. In the next chapter, this vocabulary is validated by using it in a wide variety of function modeling applications.

## CHAPTER 9. VALIDATION OF LAYER THREE: MODELING COVERAGE OF THE PHYSICS-BASED VERBS

In this chapter the proposed vocabulary of verbs is validated through modeling applications. Since the verbs are identified from review of physics phenomena and their formal definitions are composed of physics-based entities of the previous two layers, the vocabulary is first tested by modeling principles and processes of physics and the engineering sciences. Section 9.1 presents application of the energy verbs in descriptive modeling of closed systems involving multiple energy forms, without using material flows. Section 9.2 applies both energy verbs and material verbs to model existing open system phenomena from heat transfer and hydraulics. Once the vocabulary's ability to describe basic phenomena is demonstrated, it is used to model complete products in Section 9.4 (descriptive modeling) and Section 9.5 (normative modeling).

### **9.1 Coverage Testing of Energy Verbs through Descriptive Modeling (Closed Systems)**

Since the energy verbs describe energy transformation without mass transfer, only electro-mechanical processes and thermodynamic processes of closed systems can be modeled to test these verbs. Open systems with mass flow, such as thermal-hydraulic machines and principles can be modeled only for testing the material verbs. In this section, seven physical processes (Subsections 9.1.1 through 9.1.7) involving various energy subtypes are modeled using the energy verbs. These processes are chosen to address the coverage requirement that the vocabulary must support modeling phenomena

involving at least electrical, thermal, and mechanical energy forms, as mentioned in 3.1.3. For each process, a representative device is used as a typical embodiment of the process for ease of understanding and interpretation of the model. It is emphasized that the models describe the physics of the processes, rather than the devices.

The modeling approach of this exercise is descriptive (describing existing processes), rather than normative (developing an ideal process or principle). Thus, ideally, residual flows should not be captured, since a pure descriptive inspection should objectively identify the functions and flows of an observed process without cognizance of the designer's intent. In fact, as discussed in context of Figure 7.10 (Section 7.2), the identification of residues may largely vary with the application of a process in design (light bulb, heat lamp, and café lamp example). In the models below, the residual flows are identified in context of the representative device. This depiction is used at the end of Section 9.2 to explain irreversibility-based reasoning using this vocabulary.

All models presented in this exercise are qualitative, as the definitions of energy verbs presented here do not include quantitative details. For clarity, environment instances are not shown in these models. All dangling ends of flows shown are to be considered as attached to respective environment instances. Potential reasoning that can be supported with these models are also identified when applicable.

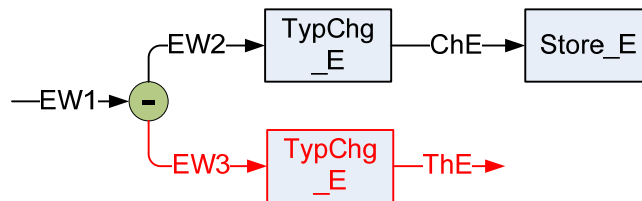
### **9.1.1 Storage and Supply of Electrical Energy (Device: Lead-Acid Battery)**

First it is recognized that many devices have multiple modes of use and often execute different functions or processes in different modes. An example is a

rechargeable battery, which can be used in two modes: charging and discharging. The function structures for these two modes are shown below. The representative device a common lead-acid battery with electrodes made of lead oxide (anode, +) and metallic lead (cathode, -) and diluted sulfuric acid as the electrolyte.

Use Mode: Charging (Storage)

The function structure of the battery during charging is shown in Figure 9.1. Energy is consumed as electrical work done by the electrical source and is spent in two accounts indicated by the balance node: (1) in overcoming the internal resistance of the electrolyte and electrodes and (2) in executing the chemical reaction where the lead-sulfate deposit and water is changed into lead, lead oxide, and sulfuric acid. The first part is lost as heat, while the electrical work EW2 is stored as chemical energy, as the resulting total energy state of the molecules of this reaction is higher than in a discharged battery.



**Figure 9.1: Storage of electrical energy as chemical potential energy**

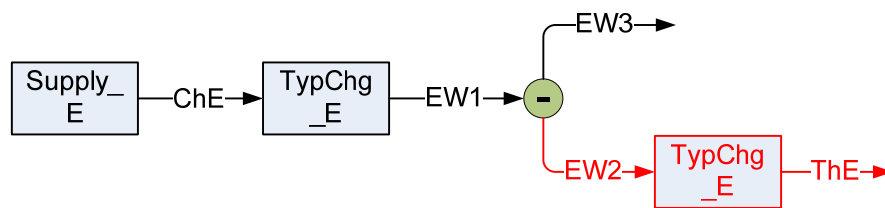
Although the main scope of this research does not include establishing a flow vocabulary for function modeling, some flows in addition to those identified in Section 8.2.1 by modifying the Functional Basis are necessary to work compatibly with the energy verbs. For example, since the physical phenomena of static electric charge



accumulation in a material body are different from those of current electricity, a distinction is necessary between these two forms of electrical energy. Hereafter, the first is modeled as electrical potential energy (EPE) and the second as electrical work (EW). Electrical work is done at all times when a current flows through a conductor. The need for this distinction will become clearer in Section 9.1.3.

Use Mode: Discharging (Supply)

The chemical reaction of charging is exactly reversed during discharging, as shown in Figure 9.2. The difference of the chemical binding energy of the reactant and product molecules is liberated as electrical work, as the products of the reaction reduce to a lower energy state. This liberation is captured in the first TypeChange\_E function. Due to irreversibility, only a part (EW3) of this total electrical work EW1 is done on the load connected to the external circuit, while the remainder EW2 is consumed to overcome internal resistance of the cell and dissipated as heat, modeled as the second TypeChange\_E function.



**Figure 9.2: Supply of electrical energy from stored chemical potential energy**

While the word supply may imply a physical flow of an entity, the Supply\_E function does not imply a “flow” of chemical energy ChE as a change of location in geometric space. Rather, it is required for derivational and topologic consistency of the

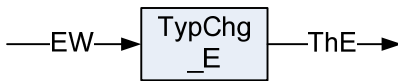
model. Without it, the flow ChE would have a dangling tail or would be inferred as entering from the environment. By definition, a function is an action performed by the device (Section 1.2.7). Here, the device is the source of this ChE and Supply\_E describes that action.

It should be mentioned that electrical work is done by the cell only when current is established by closing the external circuit. Ideally, there is a third use mode of the cell—idling—where the external circuit is open. However, once the circuit is opened, the reaction continues only momentarily until the potential difference between the electrodes builds up to create a counter electromotive force that stops the reaction and the entire electrochemical process comes to a static equilibrium, where all functions cease to exist.

### **9.1.2 Resistance to Electrical Current (Devices: Resistor, Heating Coil, Lamp Filament)**

The three basic properties of conductors that contribute to impedance to current are resistance, inductance, and capacitance [126]. These basic building blocks of electrical circuits are modeled before moving to more complex systems such as series and parallel R-L-C circuits and devices such as DC motors and DC generators.

A pure resistor offers only resistance to current, without inductance or capacitance, and electrical work spent in moving the charge against the resistance is entirely dissipated as heat according to Joule's Law [126]. The function structure for this process is shown in Figure 9.3.



**Figure 9.3: Resistive heating**

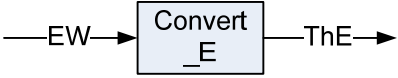
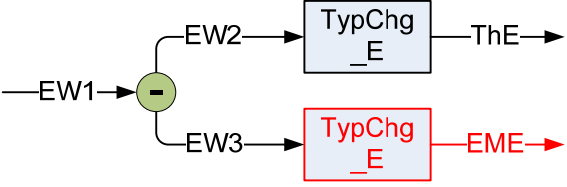
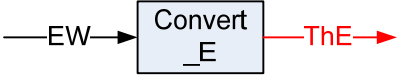
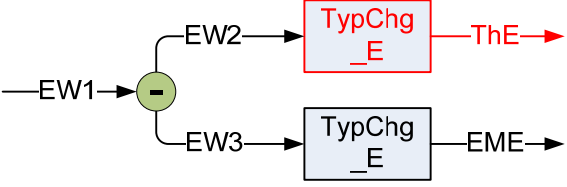
This process occurs in many devices such as electrical wires, heating elements coffee makers and hairdryers, and filaments in incandescent lamps. These devices are mentioned because they reveal certain characteristics of modeling the resistive heating process. **First**, the last two devices are usually not pure resistors, as they are formed as coils and thus produce magnetic fields around them. This is a theoretical difficulty that can be easily mitigated for this discussion by hypothetically replacing the coils with equivalent straight resistors.

**Second**, while an electric wire usually produces only heat, the heater element and lamp filament usually produce both heat (ThE) and light (EME). This light is produced through incandescence, which triggers only when the conductor's temperature exceeds a threshold, a property of the resistor material. In turn, the steady state temperature of the conductor is a function of rate of heat generation and dissipation to the surroundings, the first of which depends on design parameters such as current, and conductor properties such as conductance, length, sectional area, and melting point, while the second depends on the heat transfer modes (conduction, convection, and radiation) and their respective coefficients. Thus, whether light is produced at output ultimately depends on the quantitative details of the model, although the present modeling scope is only qualitative. To support automated reasoning about when these use modes will exist, quantitative

parameters must be captured in the representation for individual principles, an extension reserved for future extensions of this research.

**Third**, different output energy flows from the same principle can be desired or considered lost in different designs. Table 9.1 shows four representative devices of the resistive heating principle. These devices vary in design parameters such as electric current and resistance that control if light (EME) is produced and in design intent that determines if heat (ThE) is desired or not. In the two cases where multiple energy types are produced from the same function, two instances of TypeChange\_E are required, since the definition of the verb allows only one conversion per instance. These models demonstrate that the energy verbs can describe each of the cases, when accompanied by a means to mark the residual flows, which is already available in the second layer of the representation.

**Table 9.1: Model variation with design intent**

<b>Design intent</b>	<b>Heat desired</b>	 <p><b>Heating coil (coffee maker)</b></p>	 <p><b>Filament (heat lamp, hairdryer)</b></p>
	<b>Heat not desired</b>	 <p><b>Electrical wire</b></p>	 <p><b>Filament (light bulb)</b></p>
		<b>Light not produced</b>	<b>Light produced</b>
<b>Design parameters (e.g., current)</b>			

**Finally**, the representation can support modeling the same phenomenon at different spatial resolutions. For example, the production of light and radiated heat in an incandescent filament can be modeled as successive conversions from ThE to ME to EME. Light and radiated heat are electromagnetic waves with the only difference being in their frequencies. The production of these waves is a result of the atoms releasing a part of their vibrational kinetic energy (ME) as electromagnetic radiation when their temperature is elevated above a threshold [126]. Part of this radiation is described by human observers as light, as its frequency lies in the visible spectrum for humans. Thus, the identification of this energy as light (OpE) is a function of the observer being human

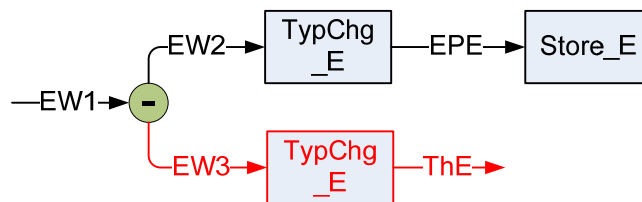
and not of the physics of the device. For this reason, the Functional Basis term optical energy is replaced with electromagnetic energy (EME) in all models in this illustration.

### 9.1.3 Storage and Supply of Electrical Energy using Capacitance (Device: Capacitor)

The storage and supply of electrical energy in a capacitor is similar to those in a lead-acid battery, with the difference that the form in which energy is stored in a capacitor is electrical rather than chemical.

#### Use Mode: Charging (Storage)

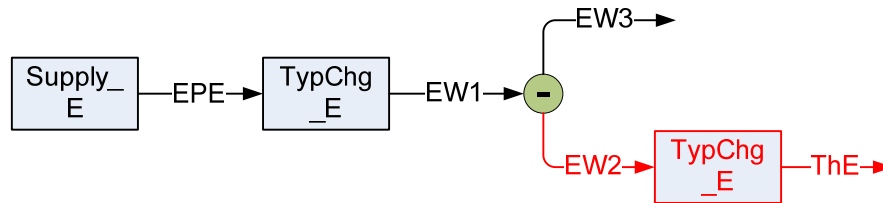
Storing charge in a capacitor requires electrical work to be done by the source, as electrons must be moved against the counter electromotive force offered by the increasingly charged electrodes. The stored form of electrical energy is the electrical potential energy EPE (static electric charge). Work spent in accumulating this charge is EW2. In addition, EW3 is electric work dissipated as heat, due to internal resistance of the capacitor is therefore the bottom TypeChange\_E function is topologically identical with Figure 9.3. EW1 is the total work spent in the process and is the sum of EW2 and EW3, as implied by the balance node.



**Figure 9.4: Storage of electrical energy as electrical potential energy (static charge)**

### Use Mode: Discharging (Supply)

The function structure for the discharge of a capacitor is shown in Figure 9.5.



**Figure 9.5: Supply of electrical work (current) from stored electrical potential energy**

Notably, the vocabulary can describe the capacitor and the rechargeable battery in topologically identical models, the only difference being the form of energy stored. This similarity indicates that as long as qualitative physics principles are considered, these two processes could be used interchangeably for storing and supplying electric energy, which is indeed true. This similarity can be also used in automated reasoning. For example, a reasoning algorithm for solution search can be written that matches the modeled functional construct with an archive of solutions where device types are mapped against functions. If this algorithm is used to seek solutions to “storage of EE as ChE” (Figure 9.1), it would possibly return different types of storage batteries, and when searched for “storage of EE as EPE”, it may return capacitors. However, based on this topological similarity, the same algorithm could return both device classes, if the search was widened as “storage of EE”, without mentioning the form of stored energy. In this manner, the representation shows potential for supporting more enhanced reasoning, which is reserved for future work.

#### **9.1.4 Production of Magnetic Field using Inductance (Device: Solenoid with or without Soft Iron Slider)**

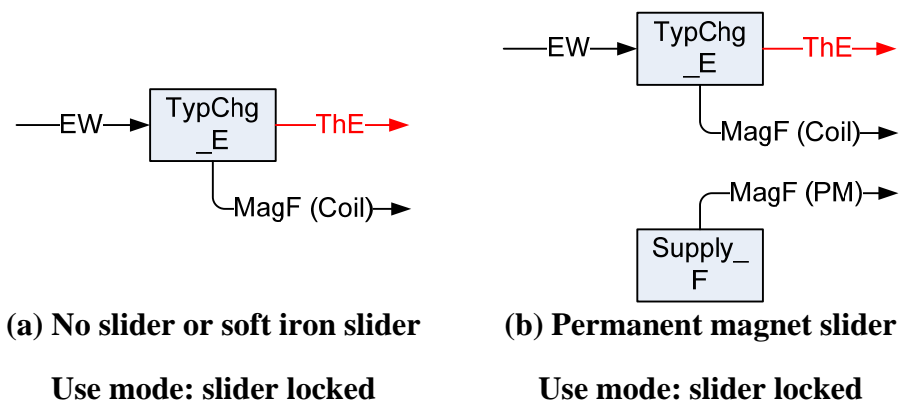
Producing magnetic fields by passing current through an inductor is a fundamental process in electrical systems and is used in applications such as transformers, motors, and generators. Basic demonstration of this principle is done by passing current through a straight conductor, which induces the magnetic field around it, while a basic engineering application is the solenoid, where current is passed through a helically coiled conductor that induces the field along the coil axis. A soft iron slider partially inserted into the solenoid is commonly used to extract linear motion (thus, work) based on this principle. Function structures for this principle are shown next, for two different use modes: slider locked and slider moving.

##### Use Mode: Slider Locked (Motion Prevented)

In this use mode, the slider does not move, either because it is locked positively or the field is not strong enough to overcome friction. Figure 9.6 shows two variants of this principle: Figure 9.6a represents the device with a magnetic but non-magnet slider, such as a soft iron core. Figure 9.6b shows the function structure for a permanent magnet slider. The distinction between these principles will become clearer in the next use mode.  $\text{MagF}$  is the magnetic field produced. No mechanical work is done, as there is no motion (locked). Thus, the entire electrical work is dissipated as resistive heat ( $\text{ThE}$ ). Figure 9.6a describes any of three processes: (1) straight conductor carrying current, (2) solenoid carrying current and no iron slider is used, (3) solenoid carrying current while soft iron slider is used but locked from moving. While mass and energy are conserved entities of



the universe, forces are not, and the topologic derivation rules do not require force balance across functions. This model is therefore consistent with the balance laws. In Figure 9.6b, the system produces two magnetic fields, one each from the solenoid and the permanent magnet, both of which are available to the surroundings. No mechanical work is done, as the slider is locked.



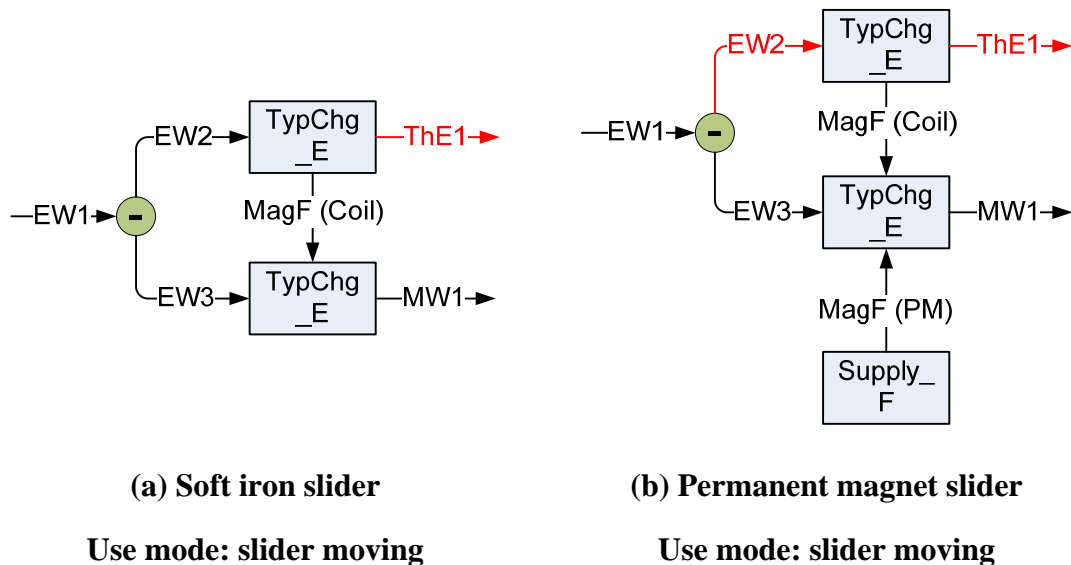
**Figure 9.6: Production of magnetic field without mechanical work by induction**

The models for resistive heating (Figure 9.3) and inductive magnetic field production (Figure 9.6) are identical except MagF. Admittedly, Figure 9.3 could be more complete as a descriptive model if the magnetic field produced around the conductor was shown. However, it was omitted as for most applications of resistive heating the field produced by a straight conductor is negligible.

#### Use Mode: Slider Moving

When the slider is unlocked and the magnetic force is strong enough to overcome friction, the slider moves, and mechanical work is done as at least the frictional resistance (force) of the slider is overcome over a distance. Figure 9.7a and Figure 9.7b show this

process for a soft iron slider and a permanent magnet slider. The total electrical work is spent in two accounts: (1) to produce mechanical work MW1 and (2) to overcome electrical resistance of the solenoid, which produces ThE1. The magnetic force from the coil is used in producing mechanical work in both cases, while the magnetic force from the permanent magnet (PM) is additionally used in the second case. Again, since forces are not conserved entities, the Supply\_F and TypeChange\_E functions do not violate conservation laws.



**Figure 9.7: Production of magnetic force from electric energy through induction**

The four models shown above use a function named Supply\_F that is not presented in the energy verbs. Similarly, the flow MagF is of the type **force**, rather than material or energy previously discussed. The reason for their use is that the role of the magnet (e.g., the permanent magnet in Figure 9.7b) is only to produce a magnetic force field, rather than to provide magnetic energy. No part of the energy required to produce

MW1 comes from the magnet, as the magnet remains in the same state after the operation is over. The only energy input to the system is through electrical current and thus, it must be a part (EW3) of the total incoming electrical work (EW1) that is spent to produce MW1. However, the current through the conductor would not produce this mechanical work unless the conductor was immersed in a magnetic field, as described by Fleming's left hand rule [126]. Thus, the magnet's role is to create a force field, which should not be described as energy, and therefore the function Supply\_F and the flow MagF are justified. However, this discussion shows that the representation could be extended to include flows of force and associated force-transforming functions in the future.

### 9.1.5 Work from Electrical Energy (Device: DC Motor with Permanent Magnet)

As the principles of electrical impedance—resistance, inductance, and capacitance—are modeled above using the vocabulary, larger electrical systems of engineering interest are modeled next. Figure 9.8 shows the function structure for the typical use mode of a DC motor.

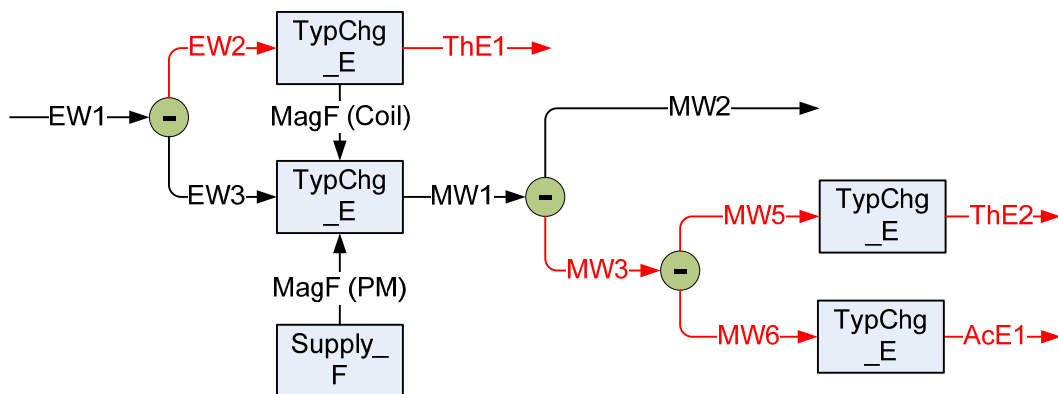


Figure 9.8: DC motor with permanent magnet

The magnet provides the field MagF (PM). Mechanical work MW2 is available at the shaft. The MW3 part of the total mechanical work produced (MW<sub>1</sub>) is consumed to maintain motion against bearing friction and is ultimately dissipated as heat (ThE2) and sound (AcE1). The other source of loss is the part of electrical work spent in overcoming the electrical resistance of the rotor coil. This portion is captured as EW2, which is dissipated as heat ThE1. This method of modeling has two major advantages in reasoning.

**First**, separation of the electrical and mechanical losses allow for calculating motor efficiency as the product of electrical and mechanical efficiencies, as shown below.

$$\eta_{\text{motor}} = \eta_e \times \eta_m = \frac{EW_3}{EW_1} \times \frac{MW_2}{EW_3} = \frac{MW_2}{EW_1} = \frac{\text{Net mechanical work output}}{\text{Total electrical work input}}$$

**Second**, this model can simulate the two other possible use modes by progressive alteration: (1) no load, where the motor freely rotates without extraction of mechanical work and (2) stalling, where the load is increased until rotation stops while the motor continues to consume electricity. In both modes, the net mechanical work output is zero. The no load mode is simulated by setting MW2 to zero, as the motor consumes only enough electric work to keep it in steady state motion against bearing friction, thus resulting:

$$\text{ThE}_2 + \text{AcE}_1 = \text{MW}_3 = \text{MW}_1 = \text{EW}_3$$

Mechanical work is produced, but is entirely consumed in overcoming friction, leaving none for net output. In stalling, motion ceases and thus  $MW_1$  should be set to zero, implying that the entire incoming electrical work must be dissipated as resistive heat from the coil, as follows.

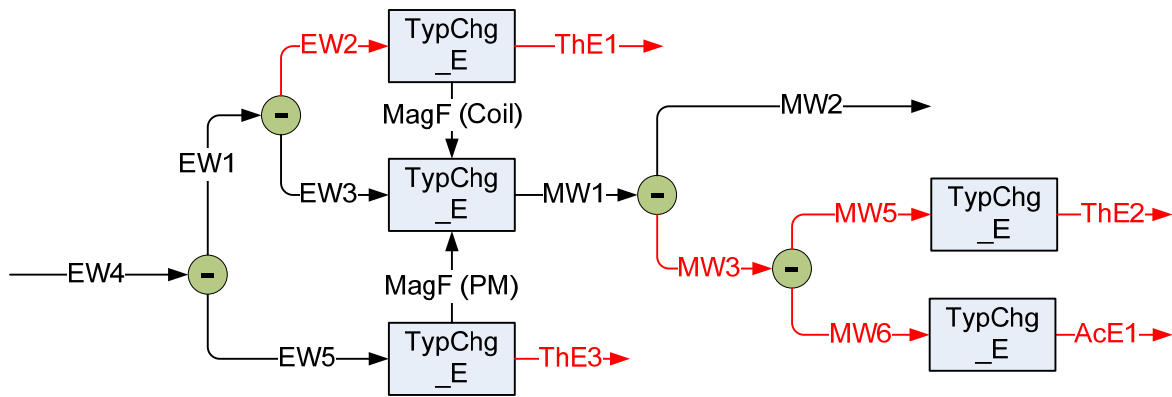
$$\begin{aligned} MW_1 &= EW_3 = 0 \\ \therefore ThE_1 &= EW_2 = EW_1 \end{aligned}$$

In stalling, there is no frictional heat or sound produced and this outcome can be reasoned as:

$$\begin{aligned} MW_1 &= 0 \\ \therefore MW_2 + MW_3 &= 0 \\ \therefore MW_2 = 0 \text{ and } MW_3 &= 0 \\ \therefore \text{both } MW_2 \text{ and } MW_3 &\text{ are non-negative} \end{aligned}$$

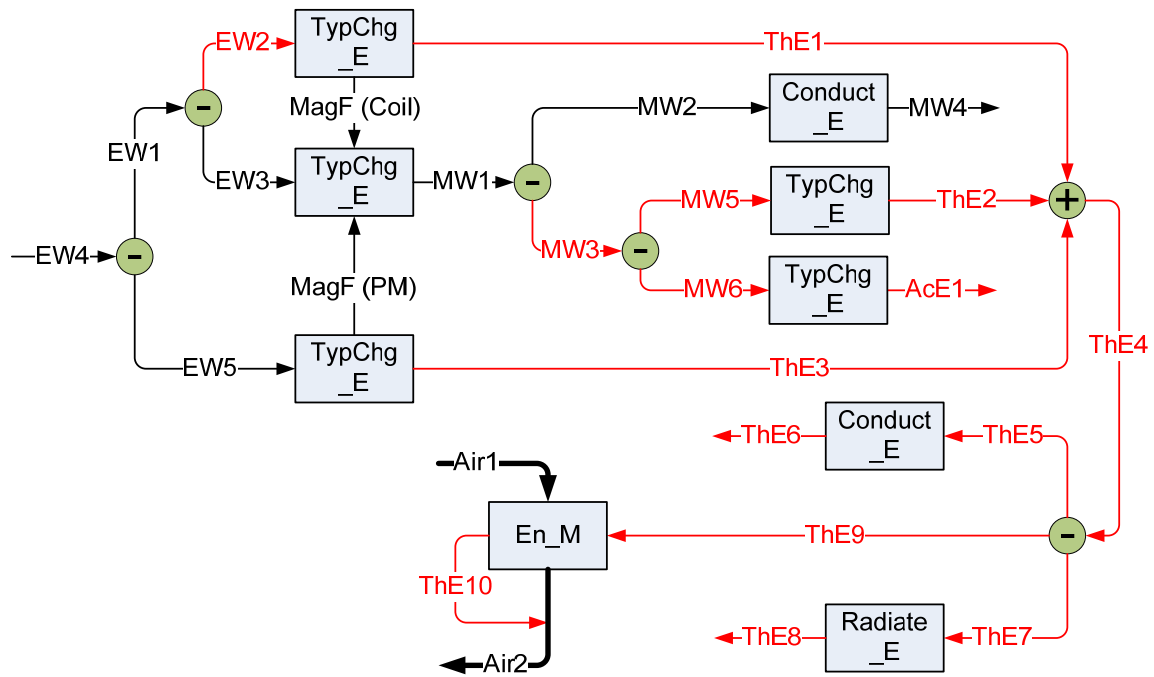
### 9.1.6 Work from Electrical Energy (Device: DC Motor with Field Winding)

The only difference in this model from the previous model is that the total incoming electrical energy must be distributed to feed the field winding and the rotor winding. This model is shown in Figure 9.9. Unlike the permanent magnet, the field winding consumes electrical work  $EW_5$ , which must be supplied in addition to the previous total input  $EW_1$ , thus increasing the total input to  $EW_4 = EW_1 + EW_5$ . The field winding is represented as a `TypeChange_E` function, instead of the `Supply_F` function in the previous model.



**Figure 9.9: DC motor with field winding**

To illustrate the use of the verbs under Transfer\_E, the model in Figure 9.9 is extended in Figure 9.10 to show the transfer of the mechanical work using the shaft and dissipation of heat to the atmosphere. Both Figure 9.9 and Figure 9.10 are correct, the difference being in functional scope (which functions are included) of the models. Figure 9.9 includes functions to produce mechanical work and heat. Figure 9.10 includes additional functions, typically executed by additional physical embodiments, such as the shaft and fins. The net mechanical work output is conducted through the shaft that acts as a conduit for work transfer and does not undergo a net displacement. MW2 is the input to the shaft, while MW4 is work available to the driven agent at the output end of the shaft. All heat produced is ultimately transferred through conduction into air, convection, and radiation, as shown with the individual functions. ThE4 implies the total heat, while the addition of heat to air (diffusion) is shown using the Energize\_M function.

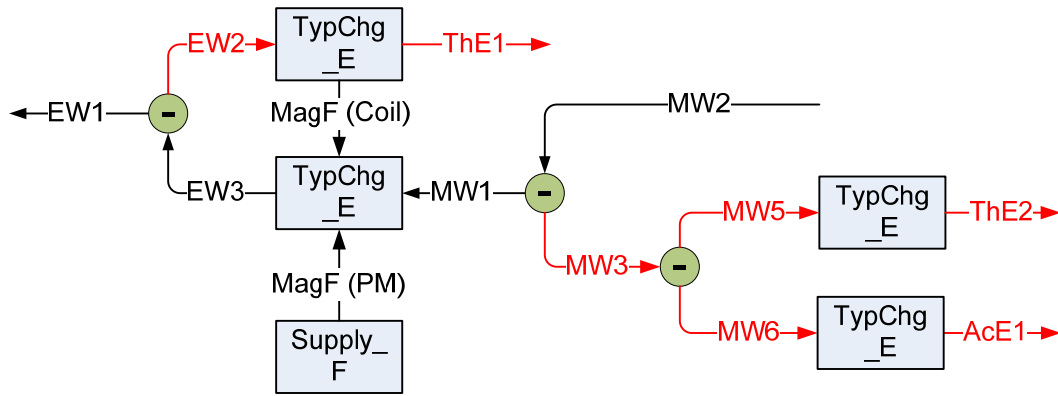


**Figure 9.10: DC motor model with energy transfer functions**

### 9.1.7 Electrical Energy from Work (Device: DC Generator with Permanent Magnet)

The converse principle of a DC motor is executed by a DC generator, where mechanical work is consumed to spin a coil inside a magnetic field, the interaction between which produces potential difference between the generator terminals. Electrical work is done only in the use mode when a load is connected between the terminals, shown in Figure 9.11. MW2 is the total mechanical work supplied by the prime mover (environment), of which MW3 is dissipated as heat and sound through friction. The remainder, MW1, is used to produce electrical work EW3 in presence of magnetic field MagF (PM). A part of this total electrical work, EW2, is consumed in overcoming the

internal resistance of the rotor coil and the counter electromotive force (back emf), and is ultimately dissipated as heat, ThE1.



**Figure 9.11: DC generator with permanent magnet**

Figure 9.11 is intentionally drawn with the energy arrows going from right to left, contrary to normal practice, in order to illustrate that this model is obtained purely by reversing the directions of the non-residual energy flows (black arrows) in the DC motor function model (Figure 9.8). The directions of the residual flows (red arrows) are not reversed. This symmetry shows that this modeling method can be used to reason that by reversing the flows through a motor, a generator could be built. Also, the non-reversal of the residual flows shows that the models are in agreement with the Second Law of Thermodynamics, since irrespective of the work flow directions, the losses always leave the system and are not recovered when the overall process is reversed.

Notably, this last model is an open system, as it shows incoming and outgoing material flows (air) and consequently, uses the material verb Energize\_M. However, modeling coverage of the material should be demonstrated through more exhaustive



modeling of basic principles of open systems with mass transfers. This demonstration is presented next.

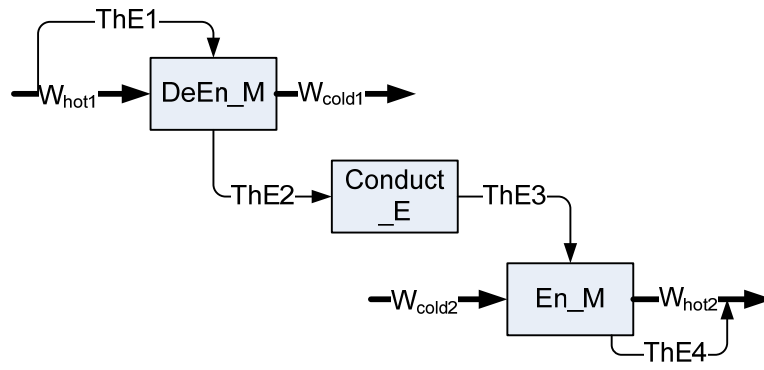
## **9.2 Coverage Testing of Material Verbs through Descriptive Modeling (Open Systems)**

Two major areas of engineering physics involving mass and energy transfer are heat transfer and hydraulics. In this section, principles and phenomena from these two areas are modeled to illustrate modeling coverage of the verbs, with special interest to the material verbs that were not tested in the previous section sufficiently.

### **9.2.1 Heat Transfer between Two fluids across a Wall (Device: Heat Exchanger Pipe)**

A basic process in heat transfer is the transfer from a hot fluid to a cold fluid separated by a conductive wall. This principle is embodied in many applications, including thermal insulations around steam pipes and hot water pipes (hot fluid inside pipe), and in evaporator coils in refrigerators (cold fluid inside). Here the representative device is a drum and tube type heat exchanger, where one fluid fills the drum and the other is passed through the tubes that run in coils inside the drum. The exchange of heat between these fluids happens through the wall of the pipe. Figure 9.12 shows a function structure for the process. The process is completed in three steps: (1) heat is diffused from the incoming hot fluid to the wall, (2) heat is conducted across the wall, and (3) heat is diffused to the cold fluid from the wall. These steps are shown by the three functions `DeEnergize_M`, `Conduct_E`, and `Energize_M`.  $W_{hot1}$  is the incoming hot fluid that loses

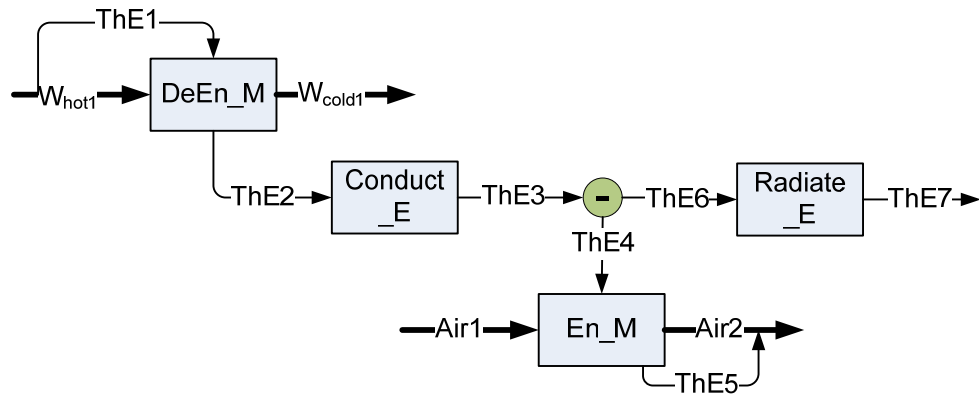
heat and exits at a different state,  $W_{\text{cold1}}$ .  $W_{\text{cold2}}$  is the incoming cold fluid that receives the heat and becomes  $W_{\text{hot2}}$ .



**Figure 9.12: Heat exchange between two fluids across a wall**

### 9.2.2 Heat Transfer from a Fluid to the Atmosphere (Device: Radiator)

An extension to the previous model is the case where the cold side of the heat exchanger is atmospheric air, as commonly applied in automotive radiators (representative device). The difference with the previous case is that on the cold side, heat is not only diffused to air, but is also radiated in space. The part diffused in air can be conducted through it or carried away by free or forced convection, but those functions are not within the scope of the model.

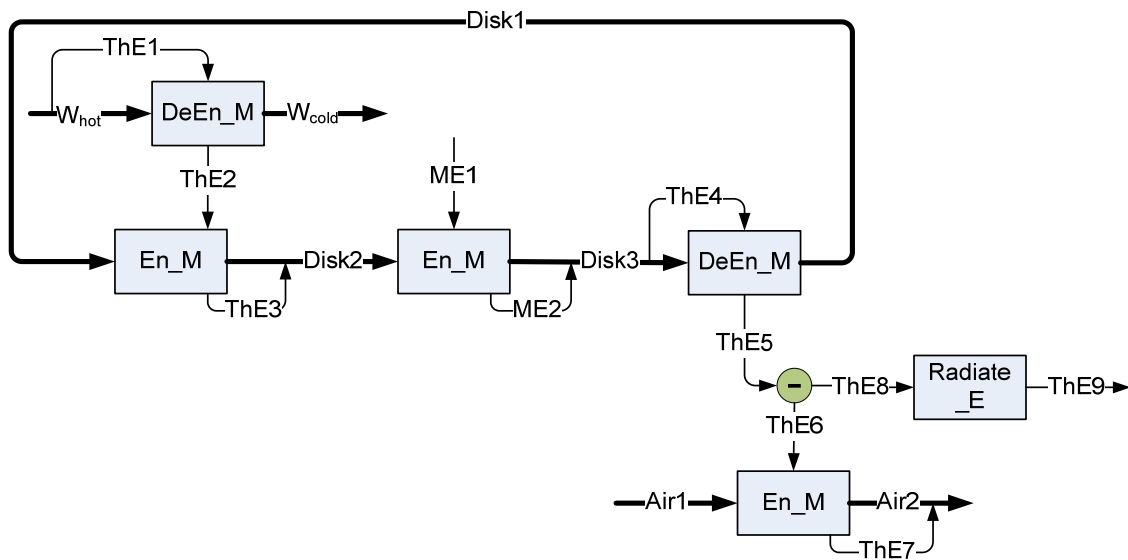


**Figure 9.13: Heat transfer from hot fluid in a pipe to atmospheric air**

### 9.2.3 Heat Transfer through an Intermediate Cycled Flow (Device: Disk Heat Exchanger)

A common application of the above principles happens in devices where an intermediate fluid (or solid) is cycled between the two temperatures as the carrier of heat from the hot to the cold fluid. The functionality of this intermediate flow is similar to that of the wall in Figure 9.12, which carries heat from the hot to the cold fluid. However, this intermediate fluid is reused and therefore energy must be expended in keeping them in motion. An example is the coolant fluid in automotive engines, which receives heat by diffusion when in thermal contact with the jackets in the cylinder block, rejects heat at the radiator, and is cycled back to the engine block using a pump. Another application is a disk type heat exchanger (representative device), as modeled in Figure 9.14. In this model, **the disk itself is modeled as the intermediate flow**. Although the disk as a whole does not flow through the system, heat is received and rejected by individual particles of the disk, which are cycled back by doing mechanical work. The material flows Disk1, Disk2, and Disk3 represent three states of a given particle of the

disk. As mentioned earlier, the definition of flow (Section 1.2.6) requires that a flow is not a part of the modeled system. However, for reused material such as the engine coolant or the disk particles, a duality of identity arises based on the system resolution. From a low resolution view, the coolant or the disk can be considered parts of the heat transfer system and the model should not depict them as a flow. In a closer look, they can be perceived as a flow. For example, when individual cooling subsystems such as the radiator or the coolant pump are modeled, the coolant needs to be treated as a flow through those systems, since it carries the energy exchanged at each step. The model in Figure 9.14 is based on this high-resolution view.



**Figure 9.14: Heat transfer using an intermediate reused flow**

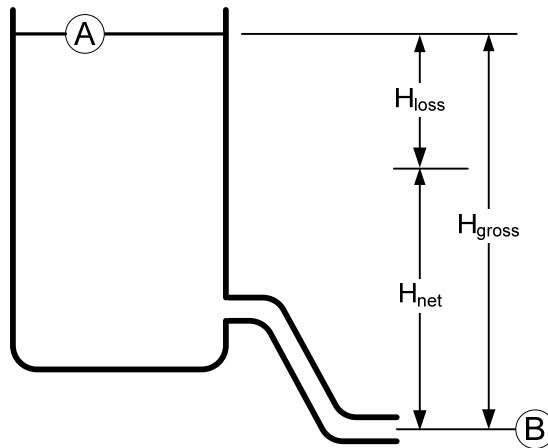
The first DeEnergize\_M function (top left) shows the loss of heat from the hot fluid  $W_{hot}$ , which is received by the particle through the first Energize\_M function (left). Here the hot fluid is de-energized and the particle is energized. The diffusion of heat

between two flows is shown with two functions, as opposed to the diffusion of heat from a fluid to the system shown with one function (DeEnergize\_M) in Figure 9.12. This two-function construct is a consequence of using the particle as a flow, rather than as a device, and illustrates the representation's ability to model both viewpoints. No Conduct\_E function is required to complete this diffusion, as used in Figure 9.12, since the two flows (hot fluid and the particle) are in direct thermal contact.

The particle is next energized with mechanical work MW2 that keeps it rotating, and comes in thermal contact with the atmospheric air on the cold side. Heat is dissipated from the hot particle to air in the same manner as described for the radiator in the previous model. The particle then returns to the original state and is reused in the same functions. No further mechanical work is added to the particle, since it is already energized with ME2. In the next subsection, some basic principles from hydraulics are modeled.

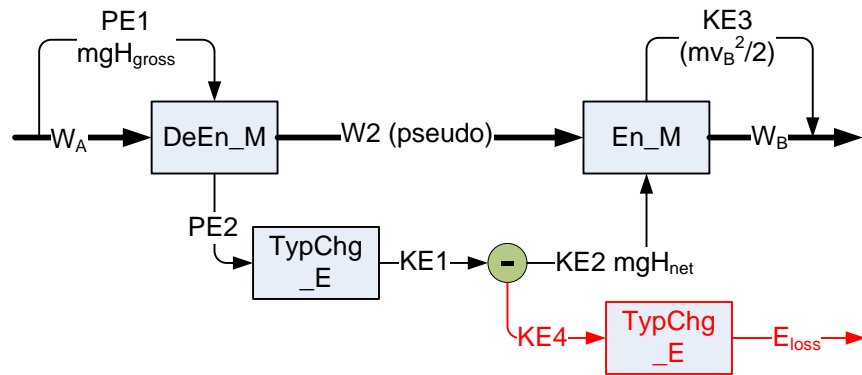
#### **9.2.4 Free Drainage of Water from a Tank (Device: Penstock of a Hydraulic Turbine)**

A basic principle in hydraulics is the drainage of water from a tank. It is applied in applications such as the penstock of a hydraulic turbine, where the reservoir is the tank and the penstock itself is the pipeline, as depicted in Figure 9.15. Gravitational potential energy stored in the elevated water is converted into kinetic energy according to Bernoulli's principle that conserves the sum of the pressure head, velocity head, and elevation head along a streamline.



**Figure 9.15: Schematic diagram of free drainage of water from a tank**

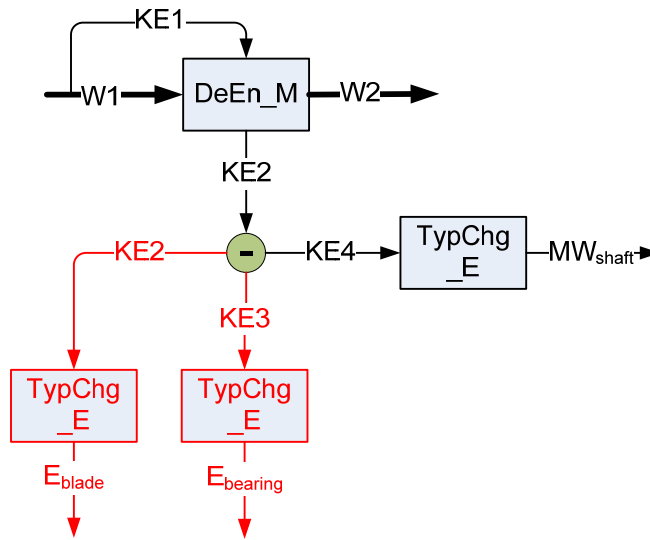
The corresponding function structure is shown in Figure 9.16. The total potential energy lost by the water in state A ( $W_A$ ) corresponds to the gravitational potential energy due to a fall through the gross head,  $H_{gross}$ . Of this total energy converted, a part is lost ( $E_{loss}$ ) in overcoming the pipe friction, while the balance ( $KE2$ ) is available at the water jet at the free end of the pipe. This energy corresponds to the net head of the system,  $H_{net}$ , and is manifested as the velocity of water in state B,  $KE3$ . The topology of this model requires the use of a `DeEnergize_M` and an `Energize_M` function, with a pseudo flow of water in the middle, since the `TypeChange_E` verb accepts only an energy flow and produces another. The vocabulary does not provide any verb for converting a carried energy flow directly into another energy flow carried by the same material. Thus, the model describes the process indirectly, where the water is first shown to be de-energized off its potential energy and later energized with the converted form, kinetic energy. The interim flow,  $W2$ , is a pseudo flow, as it is only required for maintaining continuity of the model, but does not describe any state of the water in the actual system.



**Figure 9.16: Conversion of potential energy into kinetic energy in free drainage of liquids**

### 9.2.5 Conversion of Kinetic Energy of Water to Shaft Work (Device: Francis Turbine)

A second basic hydraulic phenomenon is the extraction of shaft work from the kinetic energy of water, as performed by various types of hydraulic turbines (representative device). The function structure is shown in Figure 9.17. The water jet, as available from the penstock ( $W_B$  in Figure 9.16) transfers its kinetic energy to the blades of a Francis runner or the buckets of a Pelton wheel. This energy is partially lost, as shown using the balance node, as frictional and hydraulic losses in the blades ( $E_{blade}$ ) and also in overcoming mechanical losses such as bearing friction ( $E_{bearing}$ ). The remainder is available as shaft work  $MW_{shaft}$ .



**Figure 9.17: Extraction of shaft work from kinetic energy of water**

In summary, this chapter demonstrates modeling coverage of the vocabulary of atomic function verbs by describing a variety of existing principles of physics and engineering as function models. Principles from a variety of physics domains and engineering sciences are modeled, and a variety of flow subtypes of energy and material, in a variety of combinations are described. Although this modeling exercise could be continued to provide additional proof of coverage, **based on this preliminary examination, it is demonstrated that the proposed vocabulary provides adequate coverage for modeling existing physical and engineering principles.**

With the exception of the two topologic verbs—logical branch and logical unite—the verbs of this proposed vocabulary Chapter 8 are composed of the six entities of the first layer: Function, Source, Sink, Material, Energy, and Signal. Further, an examination of the models of this chapter reveals that the grammar rules of the first layer (section 5.2) are adhered to in these models. The grammar rules for the topologic verbs are identical



to those applicable to functions in the first layer. Thus, each model constructed here with the verbs vocabulary could as well be constructed directly from the first layer of the representation. Based on this observation, it can be said without explicit demonstration that all reasoning on conservation presented in Chapter 6 is equally applicable to these models.

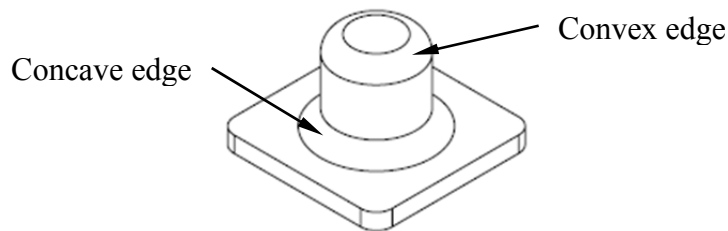
In contrast, these models are not compatible with the irreversibility algorithms of Chapter 7, since the proposed verb definitions do not include residual flows. In fact, if the algorithms of Chapter 7 were used, each model of this chapter would be identified as a violation. For example, both the DeEnergize\_M and TypeChange\_E functions in Figure 9.17 transform energy without producing residues. These errors, while detected by the irreversibility reasoning algorithms, cannot be fixed by adding a residual flow to the verbs, as was possible in the previous layers, since the definition of the verbs would not allow more than the specified number of flows. However, it must be emphasized that this violation does not imply that by extending the representation through the verbs vocabulary, the validity of the representation against the principle of irreversibility is lost. As mentioned above, the verb definitions are based entirely on the previous layers of the representation. It is only the reasoning algorithms for irreversibility that need to be rewritten to achieve compatibility with the new verbs. For example, a possible reasoning algorithm to check for residual flows for the TypeChange\_E function would be to inspect that for every instance of TypeChange\_E, another instance of TypeChange\_E exists that produces a residual flow and whose input energy flow is an output from the same logical branch node that produces the input energy to the TypeChange\_E function of concern.

For example, in Figure 9.4, the type change from EW2 to EPE is desired by the design, while the type change from EW3 to ThE is residual. This model would be accepted by the new algorithm since EW3 comes from the same balance node as EW2, indicating that a part (EW3) of the total energy spent (EW1) is lost during type changing. Similarly, in Figure 9.17, the desired conversion of KE4 to  $MW_{\text{shaft}}$  is accompanied by the two instances of TypeChange\_E that produce residual flows ( $E_{\text{blade}}$  and  $E_{\text{bearing}}$ ) and come from the same balance node as KE4. For the output side, the type change from PE2 to KE1 in Figure 9.16 is also valid, since a part of KE1 is lost as KE4. In this case, the reasoner should look for an accompanying loss from a logical branch node at the head of the output flow of the TypeChange\_E function. Thus, even with the physics-based verbs, reasoning on irreversibility can be performed with suitable new algorithms. In the next section, the verb definitions are extended to include these residual flows within the definitions, in order to enable feature-based modeling, as explained next.

### 9.3 Extension of Physics-Based Verbs with Residual Flows for Feature-Based Modeling

In this section, the verb definitions are extended to include residual flows within the definition, so that typical losses incurred during an action are automatically added to a model when these extended classes are instantiated. The motivation is to allow **feature-based modeling**, as available in geometric CAD systems [180], where one feature can instantiate a group of related entities at once. For example, a boss feature in a commercial CAD tool (Figure 9.18) includes a cylinder primitive, a Boolean unite operation between the cylinder and the base solid (plate), and fillet features at the

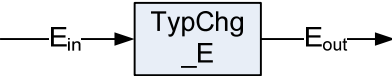
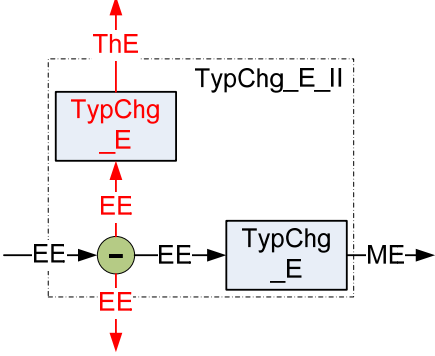
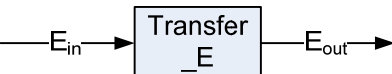
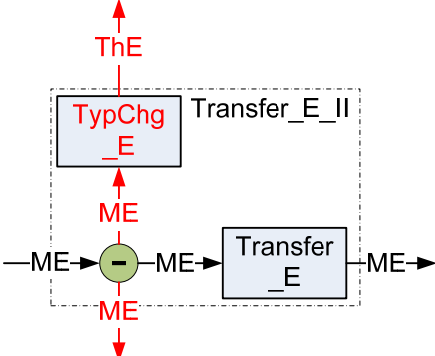
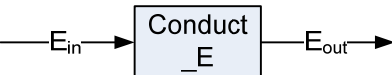
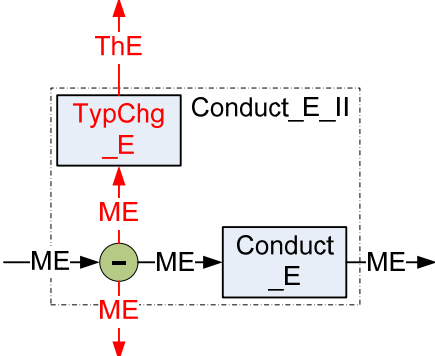
resulting edges as requested by the modeler. The fillet, in turn, consists of a sketch of the fillet cross section, a sweep of the sketch to generate the fillet solid to be added to a concave edge or removed from a convex edge, and appropriate Boolean operations to unite or subtract the fillet solid to the base solid. When a boss is added to the model, all these elementary entities of the model are instantiated in the correct order and operated through the Boolean operations to create the boss.

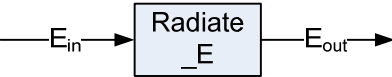
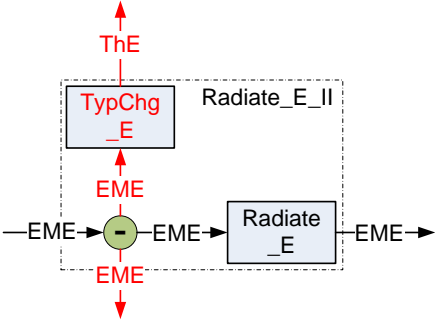
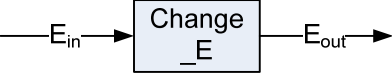
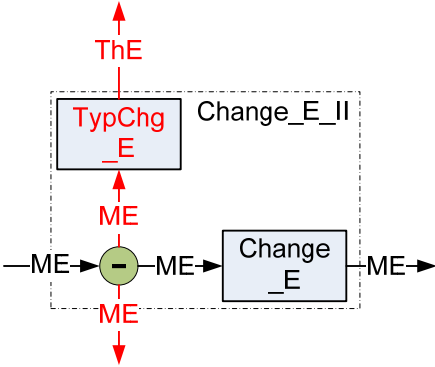
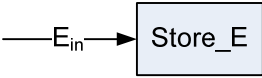
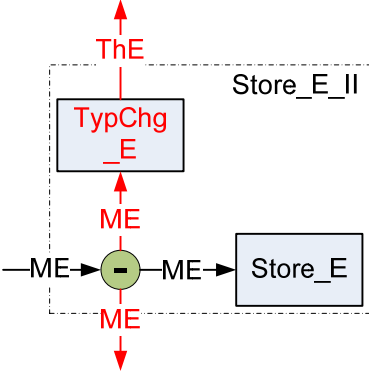


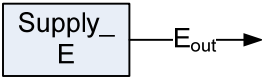
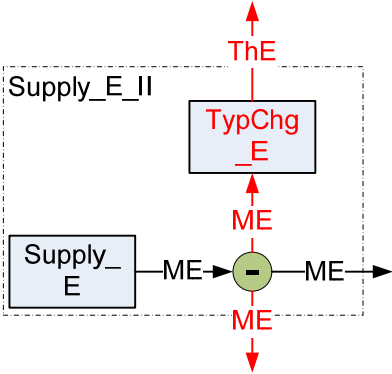
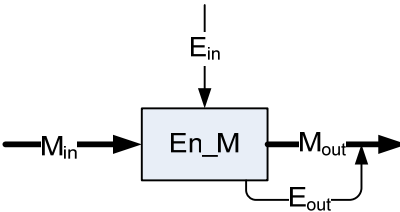
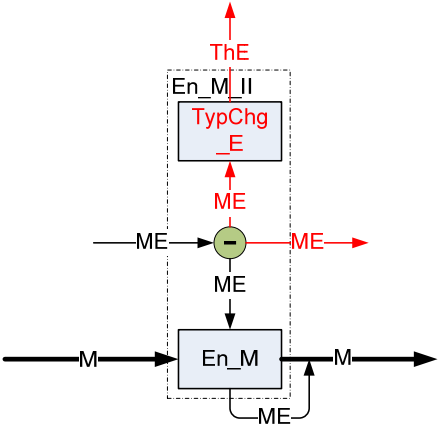
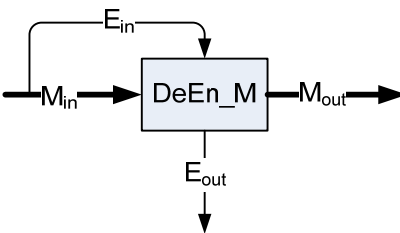
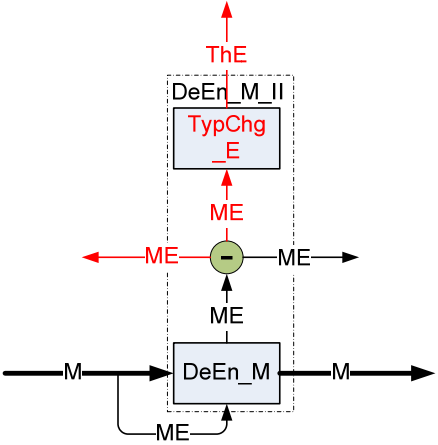
**Figure 9.18: Geometric CAD model of a boss feature**

Similarly, the intent of adding the residual flows within the verb definitions is to enable fast and easy instantiation of **function-features** that include the main function and its losses. However, since modeling requirements are difficult to foresee, similar to geometric CAD, the features may not always provide coverage over modeling situations. Low-level entities, such as the original verb set of Table 8.14, may be necessary to model non-covered actions. Table 9.2 illustrates the extensions for each physics-based verb originally presented in Table 8.14. In order to distinguish the original verbs of Table 8.14 from the extended features of Table 9.2, the former is hereafter named the **verb primitives**, while the latter is called the **verb features**, in analogy with geometric CAD primitives and features.

**Table 9.2: Extension of physics-based verbs with provisions for typical residual flows**

Verb-I	Verb-II	Verb Primitive	Verb Feature (Extension)
TypeChange_E			
	Transfer_E		
Transfer_E	Conduct_E		

Verb-I	Verb-II	Verb Primitive	Verb Feature (Extension)
	Radiate_E		
Change_E			
Store_E			

Verb-I	Verb-II	Verb Primitive	Verb Feature (Extension)
Supply_E			
Energize_M			
DeEnergize_M			

Verb-I	Verb-II	Verb Primitive	Verb Feature (Extension)
Logical_Branch			No extension
Logical_Unite			No extension

As seen from the table, the verb features entirely comprise of the primitives of Table 8.14, which are formally defined in the tables of Chapter 8 (Table 8.1 through Table 8.8). The only added information is the topological arrangement of these entities for each feature, added according to the grammar rules of Chapter 5. Thus, formal definitions of the features are not separately presented. The chain-dotted line describes the collection of primitive entities and relations that together define the feature. Each feature name is created by appending “\_II” to the primitive, for distinction.

Further, the losses for each verb are considered in three different forms, as illustrated with the three outgoing red arrows from the logical branch nodes. For example, in the DeEnergize\_M\_II feature, ME is extracted from the material flow M, one part of which is usable flow (black arrow, to the right from the balance node). The other two are lost in two ways: in the original form (ME) and after a type change (ThE). The rationale behind this default design is to provide for losses in typical applications. The

DeEnergize\_M\_II function can be used to describe a hydraulic turbine, where M is the water flow, ME is extracted by the blades and thus the DeEnergize\_M verb is the function of the runner, the black ME flow (to the right from the balance mode) is the available shaft work, the loss flow ME (to the left) is energy lost in mechanical form such as vibrations, and the loss flow ThE is energy lost from ME but after conversion into heat due to bearing friction and blade losses. Thus, three different means of losses are captured by the features.

The features include default usable and residual flows in the context of typical applications, such as the turbine for the DeEnergize\_M\_II function. It is possible that in a different application, one of the energy flows is absent or a different set of flows is chosen as the losses. Thus, during implementation, the flows should be available for deletion, addition of more flows to the balance node, and for editing their residual status. As illustrated earlier, these capabilities are available in ConMod-2q for individual flows, proving that these capabilities are realizable. Further, the use of templates does not eliminate the need for reasoning algorithms for irreversibility. Since the flows are available for editing, it is possible for a modeler to accidentally describe a function that violates the natural laws, which should be detected by the reasoner.

Based on the demonstration of coverage in Sections 9.1 and 9.2, it is expected that since the new physics-based verbs and features can describe principles and phenomena of mechanical products, they could also describe products as a whole. To this end, two types of product-level modeling are illustrated next. In Section 9.4, two models from the



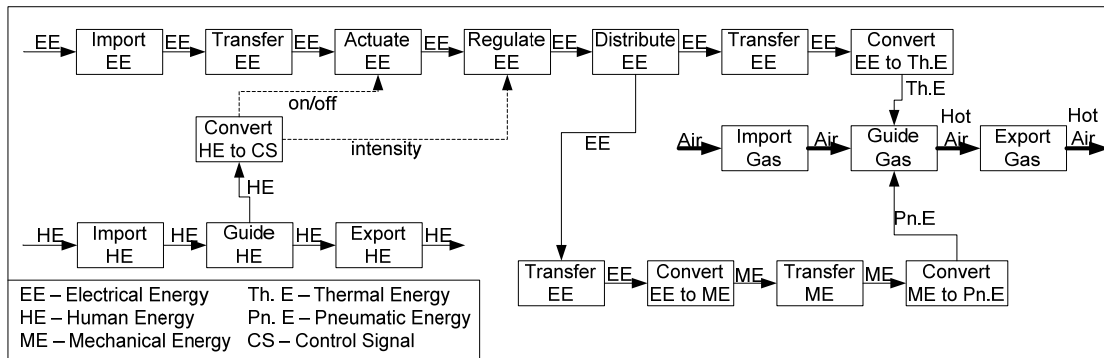
Design Repository are reconstructed using the new vocabulary in two steps: (1) using the verb primitives of Table 8.14 and (2) using the verb features of Table 9.2. Together, these two steps in Section 9.4 illustrate that the proposed vocabulary can support **descriptive modeling** of existing products. Further, in Section 9.5, the vocabulary of verbs and their extensions into the features are used to model new product concepts, as typical of an early stage design process. This exercise demonstrates that **normative modeling** of new concepts is also supported by the proposed vocabulary.

#### **9.4 Product-Level Coverage: Descriptive Models from the Design Repository**

In this section, the physics-based vocabulary of verb primitives and features are used to model products from the Design Repository, in order to illustrate that the descriptive models of existing products constructed using the Functional Basis vocabulary can be constructed using the new vocabulary and the feature set. The two products modeled are: (1) the hairdryer model from Figure 1.2 and (2) a Shop-Vac vacuum cleaner model.

##### **9.4.1 The Hairdryer Function Structure**

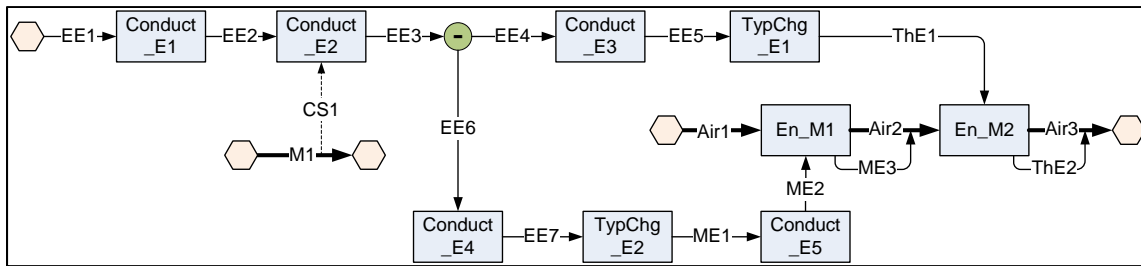
Figure 9.19 shows the hairdryer function structure of stored within the Design Repository, originally shown in Figure 1.2. The legend at bottom right indicates the flow subtypes. The rectangle around the model is the system boundary, which is crossed by incoming flows used by the device and outgoing flows produced by the device.



**Figure 9.19: Function structure model of a hairdryer stored in the Design Repository<sup>19</sup>**

Figure 9.20 shows the hairdryer model, duly reconstructed using the verb primitives. A few differences can be noticed between the two models. The boundary functions, such as Import and Export in the Design Repository model are replaced with the Environment instances in the new model. The functions at the bottom left corner of the Design Repository model, showing human interaction with the product with human energy as a flow are omitted in the new model, since from a physics-based point of view, human energy is not a basic energy type, as discussed in the discussion following Table 8.1. A control signal carried by an imported material flow M1 is used to indicate that the Conduct\_E2 function is executed in response to a signal, which replaces the Actuate EE function in the Design Repository model.

<sup>19</sup> <http://repository.designengineeringlab.org>, accessed on June 11, 2011



**Figure 9.20: The hairdryer function structure using the physics-based verb primitives**

Three points are important to note. First, the construct showing the material flow M1 connecting two environment nodes in Figure 9.20 is prohibited by the grammar rules of Chapter 5 (Construct 63, Rule 23). In this model, the role of M1 is to show that an external material is required to carry the signal CS1 that actuates the EE flow, which is a correct physical description of the device, as the human user must press a button or a switch to turn on the device. A signal has to be always carried by a material or an energy (Rule 8) and cannot be produced by a node (Rule 3), and thus the signal CS1 could not be directly imported into the mode. Yet, the human interaction is **not an action performed by the device**, and therefore, should not be described with a function, as required by the definition of functions in Section 1.2.7. Thus, M1 should not be acted upon by a function, and yet, must enter and leave the model, thus requiring the illegal construct. This example highlights that human-machine interaction is difficult to formalize through a function-based description and a richer and expressive formal representation must be used to that end. Current research on function-interaction modeling [181] begins to address these needs.

Second, the addition of the signal flow to the Conduct\_E2 function is not explicitly allowed by the definition of Conduct\_E in Table 8.14. However, as discussed in Chapter 5, a signal flow is never added to a carrier flow but can be added to any function. Since signals are not conserved entities, the conservational validity of the function is not altered by adding an incoming signal. The action of the function is altered so that the output flows are produced only when the signal is received. The Conduct\_E2 function in the model is an instance of conditional Conduct\_E, which is the function of a switch, as captured by the function Actuate EE in the Design Repository model.

Third, the function Regulate EE is not captured in the Design Repository model is not shown in the new model. The action of regulating EE, as executed by a regulator, is a transient action that shifts the system from one steady state to another. The representation captures only steady states of functions. Although the regulator changes parameters of the EE flow, it is not captured with a Change\_E function, since that change does not happen continuously for the time the device stays in a steady state.

Besides these differences, the major difference between the Design Repository model and the new model is in the constructs for adding heat and kinetic energy to the air flow. In the Design Repository model, the Guide function is used to accomplish both actions, although the function is (1) a violation of energy conservation since it has two input energy flows and no energy output and (2) a violation of its own class definition,

since the definition of Guide<sup>20</sup> [26] does not allow adding energy to material flows. In the new model, these two actions are captured by the two Energize\_M functions. Table 9.3 lists a mapping between the function names in the original model from the Design Repository (Figure 9.19) and the new model (Figure 9.20), along with typical components or subsystems used to execute each function, for ease of reference.

**Table 9.3: Function-name mapping between models (hairdryer)**

<b>Figure 9.19 Design Repository Model</b>	<b>Figure 9.20 New Model with Primitives</b>	<b>Possible Component</b>
Import EE	None – shown by the environment node	None
Transfer EE	Conduct_E1	Mains cord
Actuate EE	Conduct_E2	Switch
Regulate EE	None – transient phenomenon	Regulator
Distribute EE	Logical_Branch	Junction / solder
Transfer EE	Conduct_E3	Wire
Convert EE to ThE	TypeChange_E1	Heater coil

---

<sup>20</sup> “To direct the course of a flow (material, energy, signal) along a specific path.”

<b>Figure 9.19 Design Repository Model</b>	<b>Figure 9.20 New Model with Primitives</b>	<b>Possible Component</b>
Import HE	None – collectively shown with the flow M1 carrying CS1	These items are not functions executed by the device
Guide HE		
Export HE		
Convert HE to CS		
Import Gas	None – shown by the environment node	None
Guide Gas	Energize_M1 Energize_M2	Conduit within the barrel
Export Gas	None – shown by the environment node	None
Transfer EE (bottom)	Conduct_E4	Wire
Convert EE to ME	TypeChange_E2	Motor
Transfer ME	Conduct_E5	Shaft
Convert ME to PnE	Energize_M1	Fan rotor

Figure 9.21 shows the hairdryer model constructed using the verb features. As discussed earlier, the elements within the features are deletable and editable. In this model, the flows that do not apply for the specific application are deleted. However, the chain dotted lines defining the features are retained, in order to facilitate comparing the features retained in the model with those defined in Table 9.2. For ease of reference,

functions and flows that exist in the primitive-based model (Figure 9.20) are described with the same names in this model. Other functions and flows that are unique to this model are named without the trailing numeric. Losses coming from the individual function features are described with the main form and source in parentheses. For example, ThE (Res) is thermal energy due to resistive loss, ThE (Sur) is thermal energy lost across the surface of the device, ThE (Fric) is frictional heat loss, ME (Vib) is vibrational loss, and ME (Ac) is acoustic loss. The model illustrates that the use of features can draw the modeler’s attention to residual flows and helps to capture them in the model. Next, the Shop-Vac vacuum cleaner model with the Design Repository is reconstructed using the proposed verbs.

#### 9.4.2 The Shop-Vac Function Structure

Figure 9.22 shows the Shop-Vac function structure stored in the Design Repository, while Figure 9.23 shows the one created using the proposed verb primitives. Figure 9.24 shows the model constructed with the verb features. Table 9.4 shows a mapping between the functions names of the first two models (Design Repository model and primitives-based model).

**Table 9.4: Function-name mapping between models (Shop-Vac)**

<b>Figure 9.22 Design Repository Model</b>	<b>Figure 9.23 New Model with Primitives</b>	<b>Possible Component</b>
Import Solid-Gas Mixture	None	None

<b>Figure 9.22 Design Repository Model</b>	<b>Figure 9.23 New Model with Primitives</b>	<b>Possible Component</b>
Guide Solid-Gas Mixture	Energize_M1	Conduit
Separate Solid from Gas	DeEnergize_M1	Filter
Store Solid		Filter
Export Gas	None	Discharge port
Import Solid (Hand)	None – collectively shown with the flow M1 carrying CS1	These items are not functions executed by the device
Guide Solid (Hand)		
Export Solid (Hand)		
Import EE	None	None
Transfer EE	Conduct_E1	Power cord
Actuate EE	Conduct_E2	Switch
Convert EE to ME	TypeChange_E1	Motor
Convert ME to PnE (top)	Energize_M1	Fan rotor – suction side
Convert ME to PnE (bottom)	Energize_M2	Fan rotor – cooling side
Import Gas	None	None
Guide Gas	Energize_M2 Energize_M3	Conduit
Export Gas	None	None



<b>Figure 9.22 Design Repository Model</b>	<b>Figure 9.23 New Model with Primitives</b>	<b>Possible Component</b>
Import Human Force	None	Handle
Transmit Force	None	Handle
Export Solid (Debris)	None – different use mode	Not a function of the device

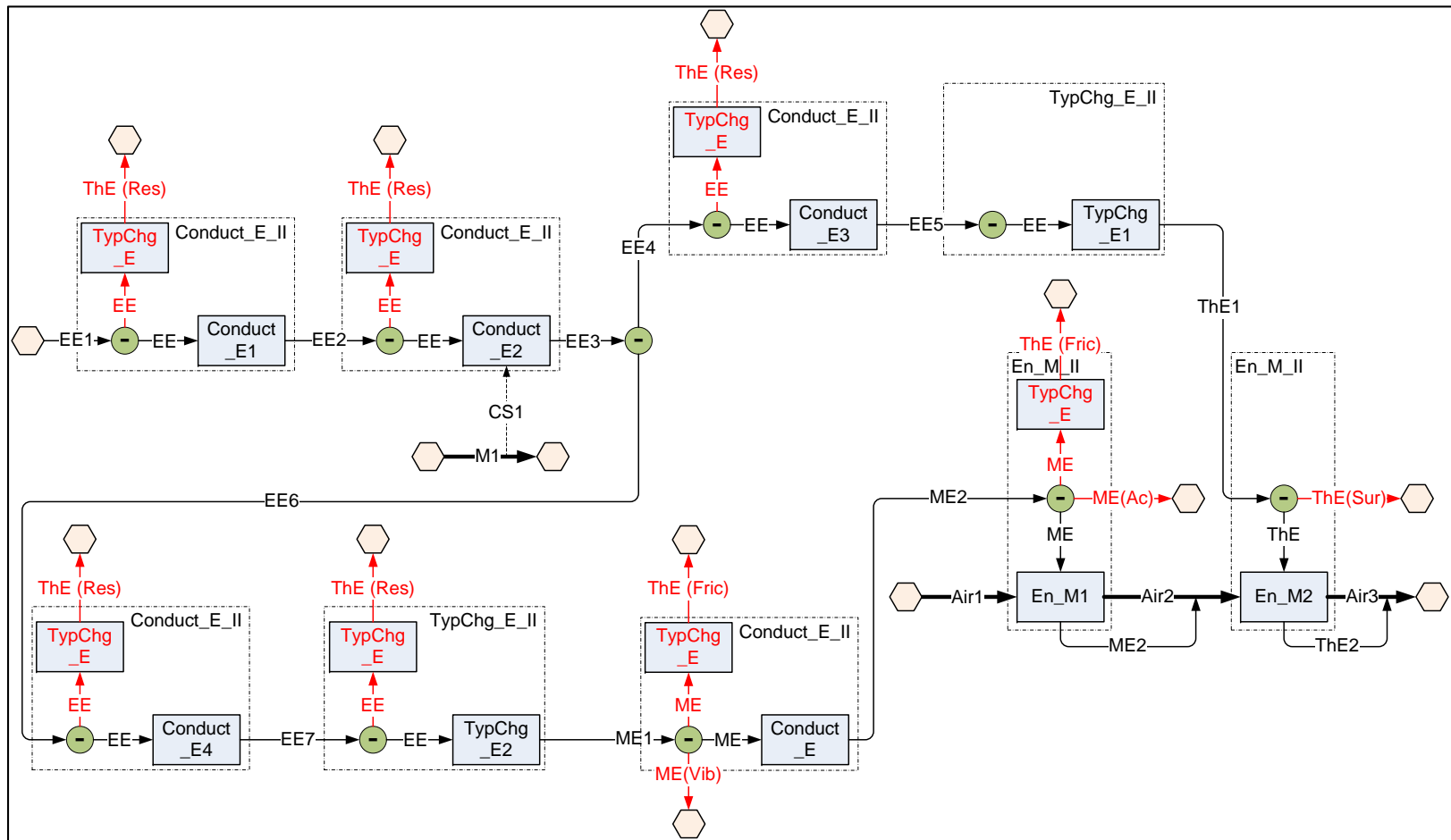


Figure 9.21: The hairdryer function structure using the physics-based verb features

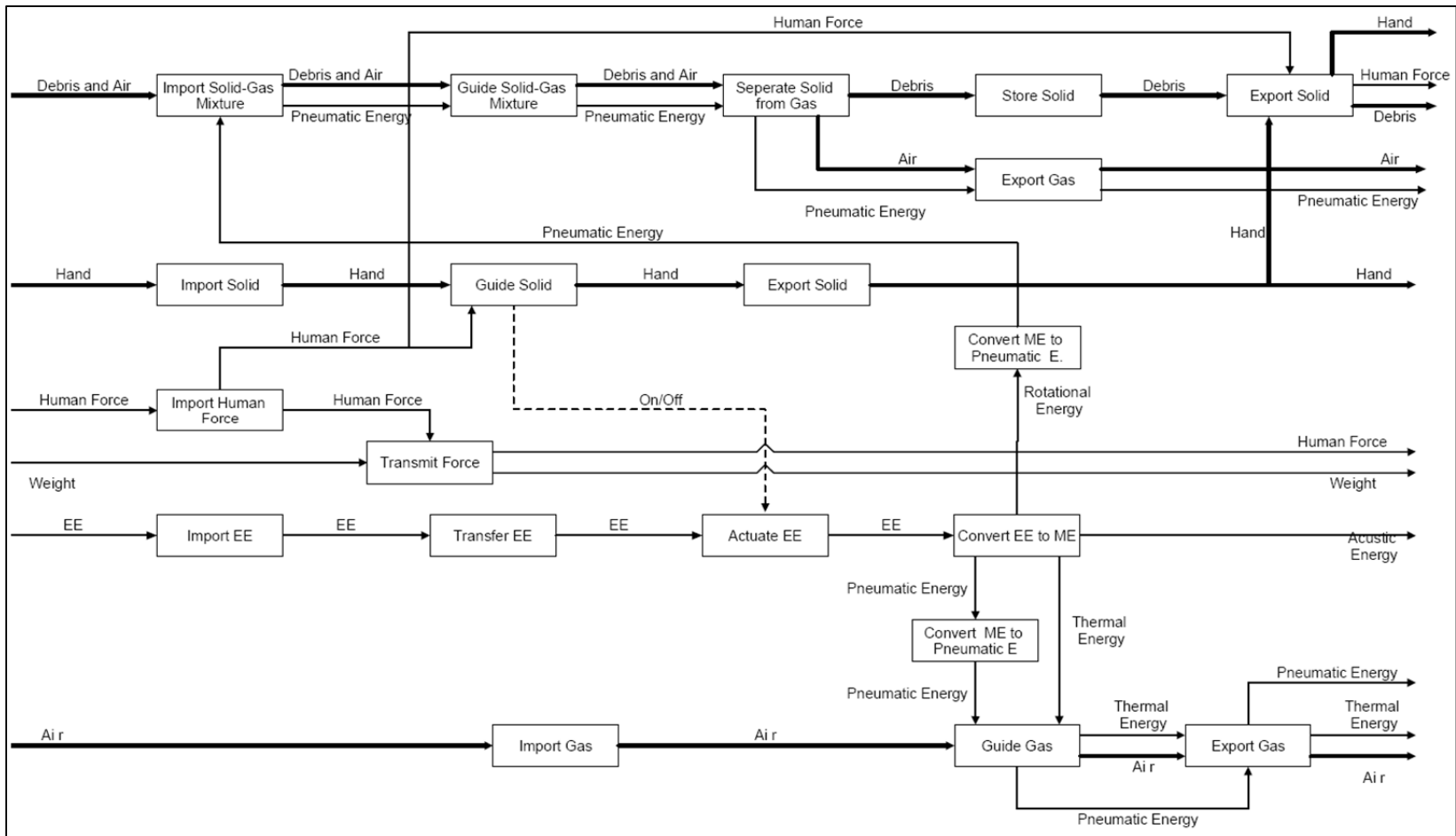
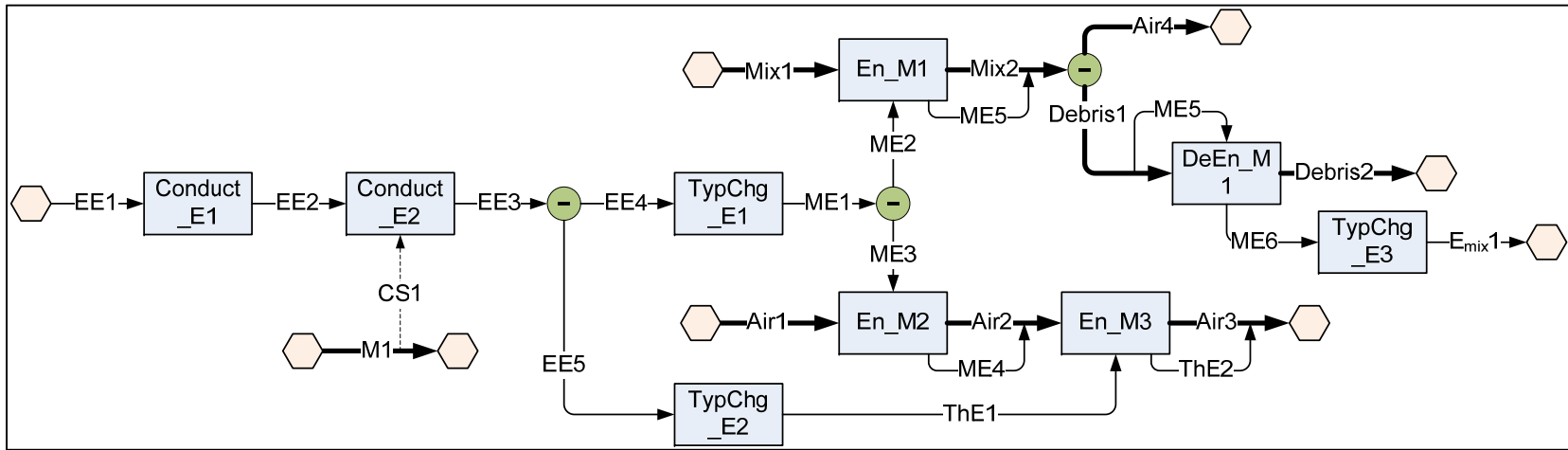


Figure 9.22: Shop-Vac function model within the Design Repository



**Figure 9.23: Shop-vac function structure using the physics-based verb primitives**

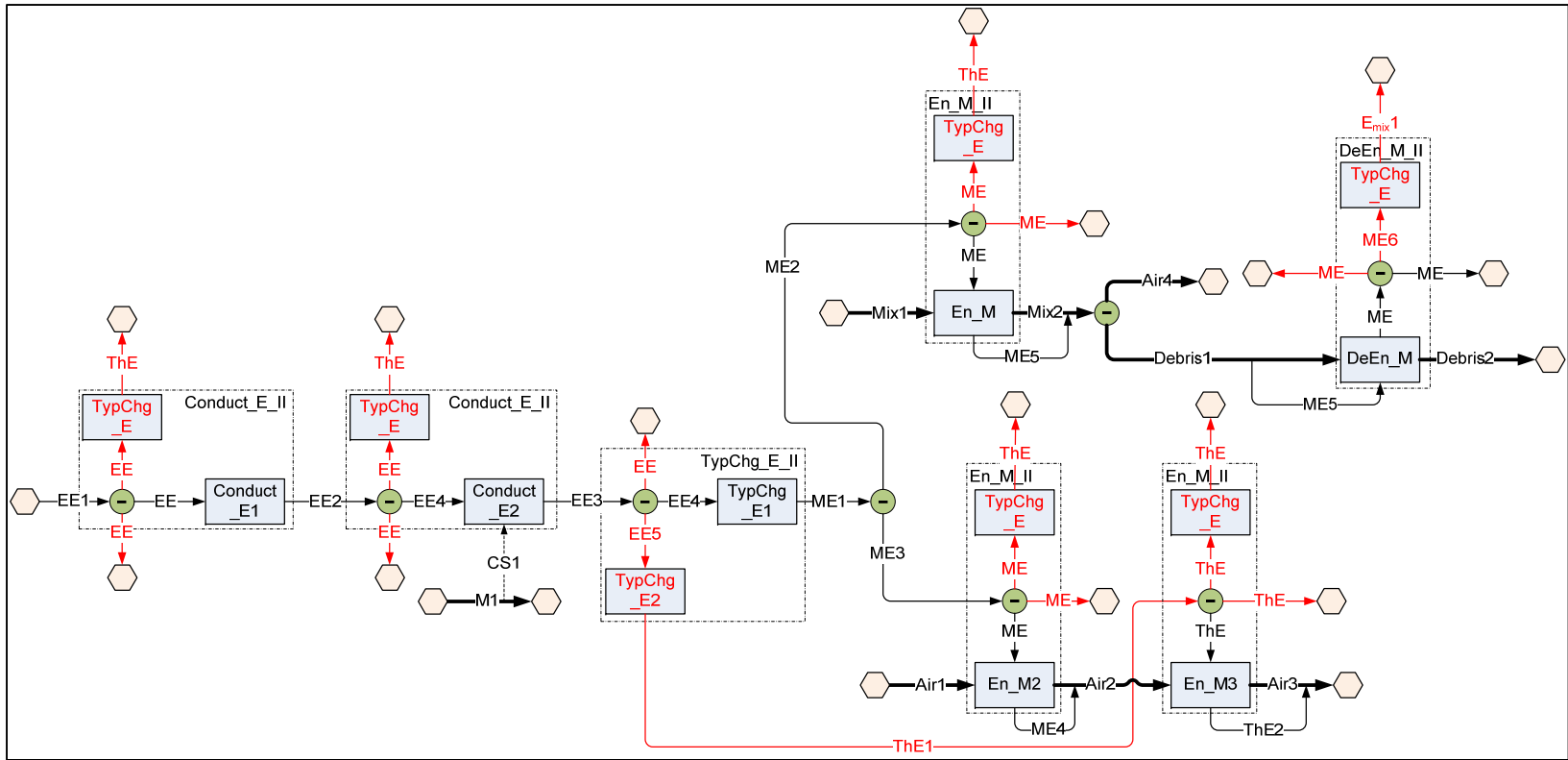


Figure 9.24: Shop-Vac function structure constructed using the physics-based verb features

Multiple use modes of the vacuum cleaner are superposed in the same model in Figure 9.22. The use-mode of emptying the stored dirt from the vacuum cleaner is represented by the last three functions in Table 9.4, while the remaining functions describe the use mode of cleaning dirt from the floor. Although this ability to superpose use-modes in a single model appears to indicate higher expressiveness of the model, the lack of a rigorous representation voids that apparent benefit. For example, terms such as Force and Human Force are not included in the vocabulary available for modeling, the branching of Human Force and Hand are shown as edge divisions without using nodes and thus violates the basic definition of graphs, the Guide Solid (hand) function lumps the action of pushing the device on the floor (the function receives human force) with the action of operating the switch (the function produces the signal to actuate EE), branches of the same Human Force flow are shown to accomplish the actions of pushing the device (Guide Solid) and emptying the dust container (Export Solid), although these actions are never performed simultaneously within the same use mode, and finally, the model does not describe that Human Force is actually carried by the Hand flow, although both Hand and Human Force are included in the model. These examples of inconsistency are results of the lack of a rigorous definition of terms and the lack of a model-level grammar constraining modeling constructs. As a result, although the model stored in the Design Repository (Figure 9.22) appears to describe the device to a human interpreter, it is unsuitable for formal reasoning.

In contrast, the new model only captures the dirt-cleaning use mode. A significant feature of this new model is its use of the Energize\_M and DeEnergize\_M

functions. Energize\_M2 and Energize\_M3 describe the cooling of the motor using a fan mounted on the motor shaft. The balance node shows the portion of the produced ME used to mobilize cooling air using this fan (Energize\_M2), while Energize\_M3 shows the addition of the produced heat to the air flow, which exits the system, thus carrying the heat away. ME2, the other part of the ME produced, energizes the air-dirt mixture (Energize\_M1), while only the debris portion of the mixture flow is slowed down to rest by the filter, thus de-energizing the dirt's kinetic energy (DeEnergize\_M1). The balance node at the head of Mix2 addresses the need for separate accounting of energy of the air and the dirt. No separate function is necessary. Thus, the functions in the old and the new model are not mapped one-to-one. Both functions: Separate Gas from Solid and Store Solid in the Design Repository model are described through the DeEnergize\_M1 function in the new model, as verified from Table 9.4.

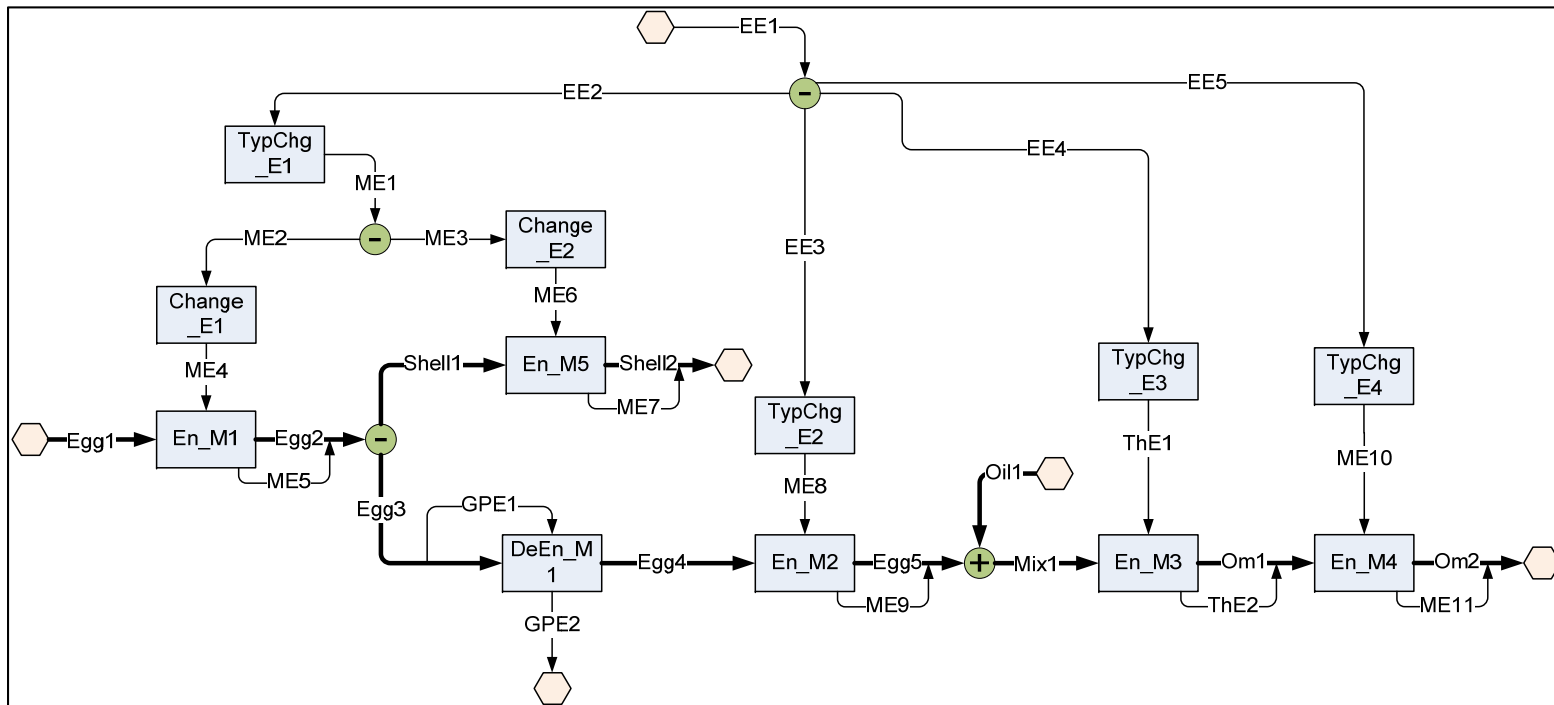
Figure 9.24 shows the model constructed with the verb features. In this model, the flows within the features are left unedited and undeleted. Similar to the feature-based model of the hairdryer (Figure 9.21), this model also repeats the flow names that appear in the primitive-based counterpart (Figure 9.23), while functions and flows unique to this model are left unnumbered. The vocabulary is next used to model a new design concept, to examine its ability to support **normative modeling**.

## **9.5 Product-Level Coverage: Normative Modeling of a New Product Concept**

The ability to support normative modeling is examined by constructing functional architectures for a design problem that is previously unsolved by the modeler. **The**

**problem is to design an automatic omelet-making machine for cafeterias that starts with raw eggs and mass produces cooked omelets by adding oil and using electrical power.** The same modeler who constructed the air heating device model during the reasoning discovery exercise in Chapter 4 is used to construct this model. A resulting functional architecture for the machine is shown in Figure 9.25. The model is constructed using the primitive verbs, rather than the features, since it is often useful to focus on only the intended functions rather than the residual functions and flows in early conceptual design [1, 2]. This model can later be examined for conservation and irreversibility and edited to include a more elaborate version with residual flows. Additionally, functions from the Transfer\_E group are not included, as the transfer of energy is a minor detail that is reserved for future iterations and elaborations of the model. Some researchers argue that avoiding auxiliary functions such as transfer makes the model more interpretable and thus useful for design communication in early stages [179].





**Figure 9.25: Normative model of an automatic omelet maker**

In this modeled architecture, an egg is cracked using mechanical energy ME4 (Energize\_M1), thus creating two parts: the shell and the egg. The shell is further energized with kinetic energy ME6, causing it to leave the system boundary. The egg is dropped, causing it to lose its gravitational potential energy GPE1. The egg is then agitated with mechanical energy ME8, oil is added to it from the environment, and heat ThE1 is then added to the mixture during the cooking process. The result is the omelet, Om1, which is further energized with kinetic energy ME10 to deliver the omelet outside the system. Each instance of ME is produced by changing the energy type from EE, in four different TypeChange\_E functions. As seen from this figure, the primitive verbs from Table 8.14 can be used to describe the concept architecture for the design.

The product-level modeling exercises (Sections 9.4 and 9.5) show that the vocabulary is better suited for describing existing devices than new ideas. The last model illustrates that a normative model **can be** created using the vocabulary, but it does not examine if the vocabulary is suitable or optimized for normative product modeling. In fact, based on modeling experience using this vocabulary, it is anticipated that physics-based verbs may not be the best vocabulary to model, interpret, or communicate new concepts, since they require the modeler to develop a concept in terms of its physics, and thus require a potentially premature commitment to physical principles, which is recommended in design texts to be reserved for later design stages [1-3]. For example, in Figure 9.25, it is difficult for a designer other than the modeler to realize that Energize\_M1 is the action of cracking the egg or that Energize\_M5 is the action of discarding the shell. To this end, notional terms proposed earlier in the top-down

vocabularies such as the Functional Basis have been used in both descriptive and normative modeling earlier. As illustrated in Chapter 3, the Functional Basis verbs are more expressive and flexible than the physics-based verbs and potentially make stronger candidates for modeling novel products. However, as illustrated in Chapter 2 and by analysis of the Shop-Vac model in Section 9.4.2, the Functional Basis verbs are defined non-rigorously and the absence of model-level grammar rules permit violating these definitions during modeling, ultimately making the models unsuitable for formal reasoning. The physics-based verbs (primitives and features) are better suited to support formal reasoning, as they are defined with entities, relations, attributes, and local grammar rules of the first two layers, which are rigorously defined, internally consistent (Section 6.1), valid against the conservation law (Section 6.2), and valid against the irreversibility principle at both qualitative and quantitative levels (Chapter 7). Thus, a tradeoff appears between the Functional Basis and the physics-based verbs: they each have their advantages and weaknesses. It would be worthwhile effort to extend the physics-based verbs and their reasoning ability to describe the notional verbs of the Functional Basis in the future, which would then support normative modeling of new products with intuitive terms for the human designer, yet would be rigorous enough to support physics-based formal reasoning.

In summary, this chapter examines modeling coverage of the physics-based verbs on physical and engineering principles, existing products, and novel products. This exercise is a main part of validating this new vocabulary. Once coverage is established, the reasoning ability of these new verbs should be examined. As mentioned at the end of

Section 9.2, since the verbs are composed of the first two layers of the representation, whose reasoning ability is already validated, explicit demonstration of reasoning with the new verbs is unnecessary. However, for illustration, the verbs should be implemented in ConMod and corresponding reasoning algorithms should be written. This exercise is reserved for future extensions.

With this chapter, the main deliverable of this dissertation—a formal representation of functions for computational reasoning—is concluded. In the next and final chapter of this dissertation, the overall outcome of this research is summarized, along with discussion on the contributions of this research to the state of the art of design automation, direct answers to the research questions and hypotheses, evaluation of the proposed representation against the requirements set in the beginning of the document, and future extensions of this work.

## CHAPTER 10. CLOSURE, ONGOING WORK, AND PATH FORWARD

This chapter is the concluding chapter of this dissertation. It summarizes the work accomplished and highlights the major contributions. The research questions are directly answered with pointers within the document where pertinent material can be found. The list of requirements set at the beginning (Section 1.3) is revisited and the proposed representation is evaluated against it, in order to show that the representation satisfies those requirements. Finally, future extensions to this research and new research questions discovered here are summarized.

### **10.1 Overall Research Outcome**

The overall research outcome is a formal representation of mechanical functions that support computer-based modeling and analytical reasoning on early design concepts and can validate them against the laws of nature. The representation is complete in three layers. The first layer (Chapter 5) identifies and formalizes the fundamental entities, relations, attributes, and local grammar rules of the Function Structure representation and is the foundational layer of the representation. The internal consistency of this layer is verified by ontological and logical examinations (Section 6.1), while its ability to support conservation-based reasoning is validated by committing the representation to the ConMod tool (Section 6.2). The second layer extends the first with three attributes (Section 7.1), which supports irreversibility-based reasoning at both qualitative level (Section 7.2) and quantitative level (Section 7.3). The third layer (Chapter 8) extends the

first two with semantic modeling and reasoning ability. Nine new physics-based verbs (seven primary verbs, two secondary verbs) are proposed and defined upon the previous two layers in syntactic and graphical form. The ability of these verbs to provide modeling coverage on physical and engineering principles, and design products, is examined in Chapter 9, thus completing the presentation of the formal representation.

## 10.2 Contributions to the State of the Art

**The most significant contribution of this research is the formalization of the graph-based Function Structure representation to support computational reasoning.** Function-based thinking and specifically function structure graphs are highly recommended in design texts as a useful means of modeling early design ideas, with the possibility of using these models to support design thinking tasks that involve reasoning of different kind. However, a tool to construct consistent function structures on the computer and perform reasoning automatically was not available until this research. Function-based reasoning, as a research topic, is studied and approached from a variety of viewpoints. Computer-based representations and tools exist in some of these areas, as outlined in Chapter 2. However, no formal representation or computer tool implementation for the graph-based Function-Structure representation existed prior to this research. Graph-drawing software tools have existed formerly at the commercial level (e.g., Microsoft Visio<sup>TM</sup>) and in design research (e.g., FunctionCAD [182]). As shown in Chapter 2, these tools can produce a visual rendering of function models, but do not have the **intelligence**—vocabulary, grammar, and algorithms—to perform any reasoning pertinent to the design itself. This research shows, by first presenting and then

implementing the proposed representation, that early design concepts can be interactively modeled and reasoned upon on the computer.

**The second contribution is addressing the area of function-based design analysis in early stages for the first time.** Automation of early design tasks, specifically design synthesis, has remained a focus of design research and computer tools that synthesize new functional concepts from a given high-level description through graph grammar-based model decomposition exist. These tools are potentially useful and credit must be assigned to these achievements, as they can further the automation of the synthesis process. For example, these tools have been extended to component-selection for concepts and providing end-to-end synthesis automation for the conceptual design stage [22]. However, none of these tools support what this dissertation defines and realizes as **reasoning**: the ability to draw inferences from a model using logic and external knowledge of the physical world, in order to discover or predict behavior of the modeled reality (concept). To this end, this research does not contradict, but complement the synthesis automation research. The design need addressed by this research is not of synthesis support. Rather, the formal representation developed in this work can be used to analyze modeled concepts for validity against the laws of nature. It is anticipated that models created through automated synthesis could be analyzed using the algorithms of the proposed representation, if the two computational systems were integrated. This integration would provide more reliable selection criteria for the synthesized concepts, as they could be validated against the natural laws during the graph grammar-based

synthesis. This extension is an interesting candidate for future exploration of this research.

**From the modeling standpoint, this research enables interactive construction of design concepts through a designer-computer mixed initiative early in the design process.** This facility replaces paper and pencil as the modeling medium with the computer, in a much similar manner of how two-dimensional geometric CAD replaced the drawing board in the early 1980's. The basic graph-drawing tools mentioned earlier also partially address this need. However, tools built upon this formal representation, such as ConMod, allow more than just **drawing** a concept; they allow the designer to engage in a **physics-based conversation** with the model while exploring ideas. The representation, and specifically its ConMod implementation, has another significant similarity with geometric CAD, in that it does not attempt to completely automate design, thus taking away the control and ownership of the design from the designer, as done in previously proposed systems [22]. The approach of this research is based on the philosophy of mixed initiative between the designer and the machine, where the designer retains control and the computer provides analytical feedback. In this sense, the representation serves as a foundation for early design CAD and forms a counterpart of geometric CAD in detail design analysis.

**From the reasoning point of view, the representation shows potential to serve as a foundational formalism for early design reasoning.** The benefits of this representation extend beyond just easy model editing on a computer and electronic



archival of design, which most computer-aided tools provide. By virtue of the underlying formal representation, these archived models could be used to support more enhanced reasoning in the future, particularly those that need multiple models, such as concept comparison for (1) complexity, (2) failure-prone-ness, (3) efficiency and other variables, or (4) manufacturability-related issues. For example, a recent research initiative [152] focuses on predicting product cost based on the complexity of function structures. This work could benefit from a tool such as ConMod in two ways. First, the compared models could be ascertained as “comparable”, if they were constructed using verbs that are at the same level of concreteness, rather than notional terms from the Functional Basis, whose concreteness potentially varies and have not been examined objectively. With notional terms, model complexity cannot be truly surrogated with count-based and connectedness-based graph metrics, since a model may contain few terms of high inherent expressiveness or many terms of low inherent expressiveness. With a physics-based vocabulary, descriptive models can be ensured to compose of verbs at comparable inherent expressiveness, since these verbs describe atomic physical actions. Second, the complexity-based algorithms could be written within ConMod to perform the complexity comparison. Similarly, a failure-based predictive tool [95, 97, 144] uses models built on notional terms from the Functional Basis. For similar concreteness-related reasons as above, this tool’s reasoning on causal propagation of failures could potentially be more objective, physics-based, and consequently reliable, if the models were constructed upon this proposed representation.

**Finally, this research provides certain benefits to design research and teaching.** Design texts commonly suggest the merit of function-based modeling and thinking in supporting early design tasks such as solution search and idea generation by human designers. However, these claims have not been objectively examined in previous research, potentially due to the difficulty in conducting human-subject studies on function modeling and analyzing their results objectively. With a formal representation now available, it is anticipated that software-based research tools to study function modeling could be built with reasonable effort. These tools could be written to automatically monitor model editing moves to serve as a protocol-gathering system in protocol studies and to analyze models post-construction, thus serving as an analysis tool in user-studies. In teaching, it is anticipated that the fundamental physics-based nature, specifically the representation's validity against the first and the second laws of thermodynamics, could be used to build software for teaching early-curricular mechanical engineering courses. For example, students in the thermal-fluid sciences could potentially examine the effect of adding an economizer or a water-preheater to a boiler plant on the efficiency of different other subsystems and the plant as a whole. In the next section, the representation is evaluated against the high-level requirements set in Section 1.3, to illustrate that the representation actually satisfies the requirements.

### **10.3 Evaluating the Representation against the High-Level Requirements**

This section revisits the high-level requirements for the representation listed in Section 1.3 and detailed in Chapter 3, and examines how each requirement is satisfied.

### Coverage over Multiple Physics Domains (Section 3.1)

It is illustrated through several modeling exercises that the representation supports modeling and reasoning on various domains of physics and mechanical engineering. Sections 9.1 and 9.2 are dedicated to test coverage over physics principles. Specifically, Section 9.1 illustrates models involving electrical energy operations, such as its storage, supply, and conversion to other forms. Section 9.2 addresses these needs for mechanical and thermal energy forms and various material forms. These material forms include (1) solids such as the disk type heat exchanger in Section 9.2.3 and the debris in the Shop-Vac model of Section 9.4.2, (2) liquids such as water in the hydraulic machines of Sections 9.2.4 and 9.2.5, and (3) gaseous such as air in the hairdryer model of Section 9.4.1. Many other examples of these energy and material types are available in the multiple models illustrated in this dissertation. **Thus, the coverage requirement is satisfied.**

### Domain-Independence of Physics Laws (Section 3.2)

The representation is fundamentally built upon the two laws of thermodynamics. The conservation principle is a consequence of the first law, while irreversibility is a consequence of the second law of thermodynamics. Thus, at a fundamental level, the representation incorporates physics laws that are universally applicable and therefore domain-independent. This character is illustrated by demonstrating the representation's ability to reason on models and detect violations of conservation (Sections 6.2 and 7.3) and irreversibility (Chapter 7). **Thus, the domain-independence requirement is satisfied.**

### Physics-Based Concreteness (Section 3.3)

The requirement is to use entities, relations, and attributes that support physics-based reasoning. The physics-based verbs proposed in the semantic layer (Chapter 8) are physics-based. They describe elementary physical phenomena that comprise mechanical systems at any larger scale, rather than notional terms such as those in the Functional Basis [26], which describe highly expressive actions in a non-rigorous manner. Each physics-based verb is syntactically defined in Chapter 8 and these definitions are adhered to in the modeling application in Chapter 9, where these terms are validated for coverage. Physics-based reasoning is already demonstrated in Chapter 6 and Chapter 7, where the ConMod application is used to detect violations of physics laws. Since the physics-based verbs are defined using these validated terms and grammar rules, their ability to support physics-based reasoning does not need to be externally examined and is deemed available. **Thus, the physics-based concreteness requirement is satisfied.**

### Normative and Descriptive Modeling (Section 3.4)

Descriptive modeling is thoroughly examined in the modeling applications of Chapter 9, where a total of 24 models are presented that describe existing physical phenomena and mechanical devices. **Thus, the descriptive modeling requirement is satisfied.** The normative modeling requirement is tested with two applications: (1) the Omelet-maker function model of Section 9.5 and (2) the air heater model constructed using ConMod in Section 6.2.3, which replicates a normative model created during the modeling exercise of Chapter 4. It is generally experienced during normative modeling using the physics-based vocabulary that the notional terms of existing vocabularies such as

the Functional Basis are potentially better suited for describing a new idea. The physics-based terms require the designer to think in terms of the concept's physics, which requires a premature commitment to solution principles. However, the first two layers do not bind the designer to use a vocabulary and supports normative modeling using free language. The air heater model of Section 6.2.3 is an example of this type of modeling, which is supported by the representation and the ConMod implementation. While the physics-based verbs can be used to model new concepts, at present the representation supports normative modeling through free language. **Thus, the normative modeling requirement is satisfied.**

#### Qualitative Modeling and Reasoning Support (Section 3.5)

The ConMod models presented to illustrate conservation-based reasoning (Section 6.2) and qualitative irreversibility reasoning (Section 7.2) are all qualitative and do not include numeric data. Further, the 25 models used to illustrate modeling coverage of the physics-based verbs in Chapter 9 are all qualitative. Qualitative reasoning on conservation and irreversibility are shown in Section 6.2 and Section 7.2 respectively. **Thus, the qualitative modeling and reasoning requirement is satisfied.**

#### Extendibility (Section 3.6)

Throughout the dissertation, the representation is built by extending upon previously presented layers. Further, the extension into supporting quantitative reasoning is explicitly illustrated by implementing quantitative reasoning in ConMod in Section 7.3. Quantitative reasoning on these models to compute different unknown variables such as

power required, efficiency, the power of a flow, or other physical parameters of the functions or flows would require integrating this representation with a variational solver, as indicated in Section 7.3. Complete development of this extension is reserved for future work. Further, the possible extension of the representation to formalize notional verbs from existing top-down vocabularies or free language is illustrated in Section 10.5.1 and the possibility to support causal reasoning is illustrated in Section 10.5.3.

**Thus, the extendibility requirement is satisfied by the representation.**

#### Scalability (Section 3.7)

Throughout the dissertation, models of a wide range of size and connectedness are illustrated. In terms of physical size of the concepts, the models describe artifacts as small as a spring or the electrodes of a lead-acid battery, where the phenomenon is conversion of EE to ChE (Section 9.1.1). At the other extreme, entire hydraulic turbines are modeled (Section 9.2.5). However, the scalability concern of the representation does not arise from the physical size of the artifacts, rather the complexity of the model related to the number of instances and their connections. The smallest models in this dissertation are the Store\_E and Supply\_E functions in Section 8.2.1, each of which has only one function and one flow. Starting from this size, models of a variety of sizes are presented. One illustration of scaling is provided in Section 6.2.3, where a complete product is modeled in ConMod. The concern for scaling arises from the ability for the computer data structure to hold information for large models and support executing the algorithms. As explained in Section 3.6, the Big-O complexity of these algorithms does not raise significant concern that modern computer hardware would run out of resources for

models in ConMod. **Thus, although the scaling requirement is not formally tested, it is anticipated that this requirement is satisfied.**

With this discussion, the proposed representation is checked against the requirements set at the beginning of the dissertation. The next section provides answers to the research questions and summarizes the test results of the research hypotheses.

#### **10.4 Answers to Research Questions and Hypotheses**

In this section, the research questions from Chapter 1 are answered and the hypotheses are concluded. The questions are answered starting from the sub-questions, leading up to the answer to the respective main questions. Table 10.1 summarizes the answers.

**Table 10.1: Answers to Research Questions and Hypotheses**

**RQ-1.1      What specific physics-based analytical tasks should be supported?**

**Task 1      Reasoning Discovery**

**Answer**      The twelve reasoning needs in the three categories discovered through the modeling exercise in Chapter 4 and summarized in Table 4.15 are supported. The last two items in this list, related to resolution-based reasoning, are not supported due to scope limitations. The complete list of supported reasoning includes redundant function, dangling head, dangling tail, barren flow, orphan flow, one-in-many-out derivation, many-in-one-out derivation, material transformation without energy, detecting missing residual flows, and power required. In addition, an eleventh derivation of

the type many-in-many-out is supported, as mentioned in Table 6.14.

**RQ-1.2**      **Are these reasoning tasks algorithmically solvable?**

**Task 2**      **Algorithmic Deduction**

**Answer**      The algorithm for all twelve reasoning types identified in Table 4.15 are presented in Chapter 4 (Algorithm 4.1 through Algorithm 4.12) and also implemented in ConMod (Chapter 6 and Chapter 7). In addition, although the algorithm pseudo code for the additional reasoning in Table 6.14 is not presented, the algorithm is implemented in ConMod and illustrated in Chapter 6. Thus, all reasoning needs identified here are algorithmically realizable.

**RQ-1.3**      **What information elements must be captured to support the algorithms?**

**Task 3**      **Information Extraction**

**Answer**      The information elements necessary for the representation are those identified by systematically inspecting the algorithm pseudo codes of Section 4.2 and listing them in Table 4.16. In total, there are 22 elements of information, including six entities, six relations, and ten attributes.

**RQ-1.4**      **Is the representation internally consistent?**

**Task 4**      **Consistency Verification**



**Answer** Yes. Internal consistency of the representation is examined in three sections within Chapter 6:

Logical Examination of Exhaustiveness of Local Grammar (Section 6.1.1),

Logical Examination of Consistency of Local Grammar (Section 6.1.2),

and

Ontological Examination of Consistency of the Vocabulary (Section 6.1.3).

**RQ-1.5** **Can the representation support physics-based reasoning in early design?**

**Task 5** **Validation of Conservation**

**Answer** Yes. The ability to support conservation-based reasoning is demonstrated by implementing the representation in the ConMod software tool, in two steps:

Demonstration of External Validity against Conservation Laws (Section 6.2) and

Implementation and Validation: Quantitative Irreversibility Reasoning (Section 7.3)

**Task 6** **Validation of Irreversibility**

**Answer** Yes. The ability to support irreversibility-based reasoning is demonstrated by implementing the representation in the ConMod software tool, in two steps:

Implementation and Validation: Qualitative Irreversibility Reasoning  
(Section 7.2) and

Implementation and Validation: Quantitative Irreversibility Reasoning  
(Section 7.3)

Based on the answers to the above sub-questions, RQ-1 is answered next.

**RQ-1**      **What are the entities, relations, attributes, and grammar rules necessary to formalize the Function Structure representation, in order to support (1) consistent models and (2) analytical computational reasoning on concepts based on conservation and irreversibility?**

**Answer**      The entities, relations, attributes, and grammar rules necessary to formalize the Function Structure representation are the ones presented in Chapter 5 and Chapter 7. Specifically the entities of Table 5.1 and Figure 5.1, the relations of Table 5.12, and the attributes of Table 5.13 support conservation-based reasoning, as shown in Section 6.2. Further, the three additional attributes of Table 7.1 and Figure 7.2 support irreversibility-based reasoning illustrated in Sections 7.2 and 7.3.

**RQ-2.1**      **Does the proposed verb set provide modeling coverage over a variety of physics and mechanical engineering principles and devices?**

**Task 7**      **Modeling Coverage Testing**

**Answer** Yes. The coverage of the proposed physics-based vocabulary on the principles and devices illustrated through model construction applications in Chapter 9. Specifically, Section 9.1 illustrates coverage over processes in closed systems without mass transfer. These processes involve various energy forms including electrical, mechanical, and thermal. Section 9.2 extends this coverage testing on open systems with both material and energy flows. In all, seventeen models are used to illustrate this coverage.

**RQ-2.2 Can it support consistent descriptive modeling of existing devices?**

**Task 8 Descriptive Modeling**

**Answer** Yes. The coverage over descriptive modeling of existing mechanical engineering devices is shown in Section 9.4, using two models: a hairdryer and a vacuum cleaner from the Design Repository database.

**RQ-2.3 Can it support consistent normative modeling of new design concepts?**

**Task 9 Normative Modeling**

**Answer** Yes. The coverage over normative modeling is shown through modeling two new concepts. The first is the omelet-maker machine of Section 9.5. The second is the room air-heater device, originally modeled as a new product by a designer in the modeling exercise of Chapter 4 and later replicated in ConMod in Section 6.2.3.

However, it should be mentioned that the physics-based verbs are generally

found to be less useful for normative modeling than the notional terms of the top-down vocabularies such as the Functional Basis.

Based on the answers to the above sub-questions, RQ-2 is answered next.

**RQ-2**      **At the physics-based concreteness level, is there a finite set of verbs that can describe a variety of physical phenomena and mechanical engineering principles as functions?**

**Answer**      Yes. The eleven verbs proposed in Chapter 8 can serve this purpose. Specifically, there are seven energy verbs in a two-level taxonomy (five primary verbs: Section 8.2.1 and two secondary verbs: Section 8.2.2), two material verbs (Section 8.2.3) and two topologic verbs (Section 8.2.4) in this vocabulary.

The research hypotheses related to the primary research questions can now be answered.

**RH-1**      **The entities, relations, and attributes shown in Figure 5.1 and the grammar rules of Section 5.2 can support consistent modeling and conservation-based reasoning on concepts.**

**Status**      TRUE.

The hypothesis is tested by applying the entities, relations, and attributes

from Figure 5.1 and Table 5.2, and the grammar rules of Section 5.2 in constructing function structure models. To illustrate conservation-based reasoning, these information elements are implemented in the ConMod software tool in Section 6.2, where specific reasoning tasks under the conservation type are individually demonstrated. Finally, a product-level modeling and reasoning illustration is also performed in Section 6.2.3.

**RH-2      The representation shown in Figure 7.2, including the grammar rules of Section 5.2, can support irreversibility-based reasoning on concepts.**

**Status      TRUE.**

The hypothesis is tested by applying the entities, relations, and attributes from Figure 5.1 and Table 5.2, the grammar rules of Section 5.2, and the additional three attributes highlighted in Figure 7.2 in constructing function structure models. To illustrate irreversibility-based reasoning, these elements are implemented in the ConMod software tool in Section Chapter 7, where specific reasoning tasks under the irreversibility type are individually demonstrated. Both qualitative and quantitative reasoning on conservation and irreversibility are demonstrated in this chapter.

**RH-3      The eleven verbs presented in Chapter 8 (Table 8.14) can describe principles from physics and mechanical engineering involving electrical, mechanical, and thermal energy.**

**Status** TRUE.

This hypothesis is tested by applying these verbs to model principles from physics and mechanical engineering in Chapter 9. Specifically, Section 9.1 illustrates coverage over processes in closed systems without mass transfer. These processes involve various energy forms including electrical, mechanical, and thermal. Section 9.2 extends this coverage testing on open systems with both material and energy flows. In all, seventeen models are used to illustrate this coverage.

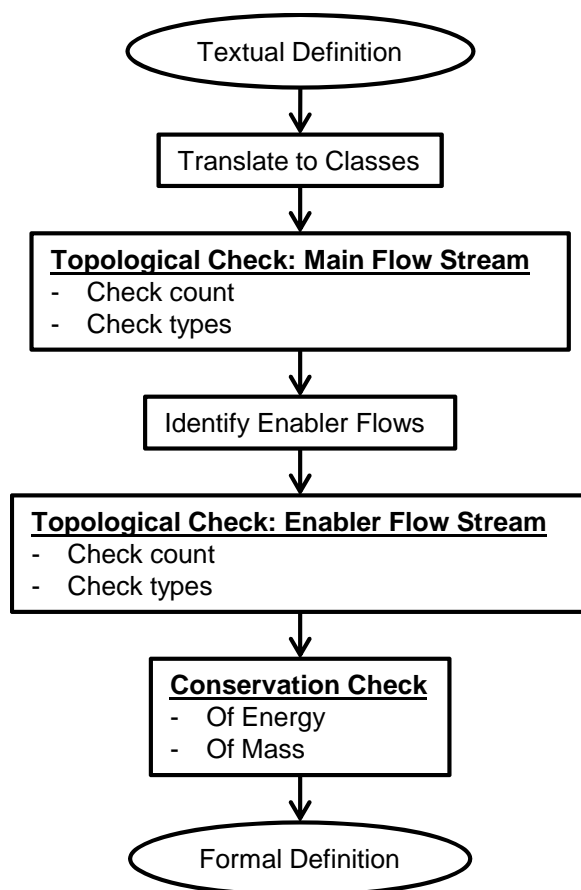
The above discussion concludes the answers to the research questions and hypotheses. Overall, it is observed that the representation satisfies the high-level requirement set in the beginning of the document and the research hypotheses are found to be true. One overarching merit of this work is that the representation and its reasoning are both verified and validated within the dissertation by logical and ontological examinations and live demonstration through software code. However, as all research should do, this work raises new research questions and directions for exploration in the near future, as discussed in the next and final section of this dissertation.

## **10.5 Ongoing Extensions and Future Research Directions**

Two immediate future research directions emanating from this dissertation—formalization of notional verbs and designer usability examination—are already underway and are described in Sections 10.5.1 and 10.5.2. Additional future research directions are summarized in the last two subsections.

### **10.5.1 Ongoing Work: Formalization of Notional Verbs**

It is identified through the normative modeling exercise that the notional terms are better suited for new product description than the physics-based verbs due to their higher flexibility and expressive power. Conversely, the physics-based verbs are more rigorous and formally reason-able than the notional verbs. It would be useful to merge the benefits of both of these approaches into a unified set of verb definitions. Work toward formalizing the notional terms is already in progress [48]. As a first step, a protocol is proposed that can be used to identify necessary physics-based components for formalizing the textual definition of a given notional verb from any vocabulary or from the free language. This protocol is shown in Figure 10.1.



**Figure 10.1: Protocol to formalize notional verbs (In-progress future work)**

The protocol requires first critically examining the textual definition and translates the intent of the text into a formal class using set-based first-order logic statements. Then, it ensures types and counts of incoming and outgoing flows for the verb. Some verbs need enabler flows of material (e.g., catalysts) or energy (e.g., detonators) for executing their complete notional action. These flows are identified. Finally, the expressions for conservation of mass and energy are explicitly written and residual flows are identified to ensure conformity to irreversibility. This protocol is illustrated here by formalizing the verb Branch from the Functional Basis vocabulary.



Branch is defined in the Functional Basis as: “To cause a flow (material, energy, signal) to no longer be joined or mixed” [26]. This definition suggests by use of the term “a flow” that the verb receives only one input flow. Although the common meaning of the word Branch implies multiple output flows, in the absence of explicit declaration, it has to be “literally” inferred from the definition that it produces only one output flow. The input flow is mentioned to be of any sub-type within the classes Material (M), Energy (E), and Signal (S), while the output flow cannot be of the types Mixed or Joined. The resulting translation in first order logic statements is presented in Figure 10.2. The keyword **Type** is a method that returns the class name of its calling instance. Next, this definition is subjected to the four checks.

```

Class : Branch{In_List, Out_List}
{
    In_List = {I1}
    Out_List = {O1}
    I1.Type ⊆ (M ∪ E ∪ S)
    O1.Type ⊆ (M ∪ E ∪ S) - (Mixture ∪ Joined)
}

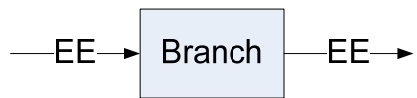
```

**Figure 10.2: Syntactic translation of the existing textual definition of Branch available in Functional Basis literature [26]**

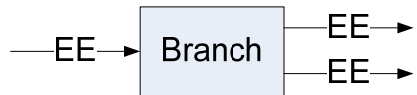
#### Topological Check of the Main Flow Stream in “Branch”

There are two lists of flows, *In\_List* and *Out\_List* in the main flows stream of Figure 10.2, where the following inconsistencies can be observed.

Count check: Since the definition “literally” allows only one output flow, it conflicts with the intent of the verb. For example, according to the definition, the model in Figure 10.3 is accepted by the definition, although the function does not actually branch the flow. Conversely, Figure 10.4 is not acceptable by the definition, although it actually represents the branching action. Therefore, the new formalized definition must allow for multiple output flows.

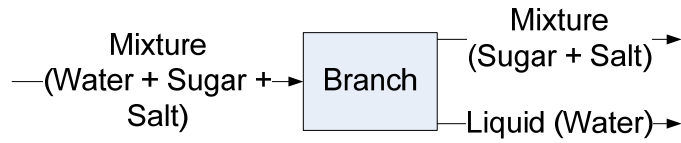


**Figure 10.3: Incorrect use of the verb Branch allowed by its definition**



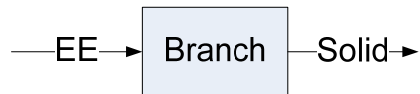
**Figure 10.4: Intended usage of the verb Branch not supported by its definition**

Type check: The term Joined is mentioned in the definition, but is not a Functional Basis noun type and therefore should be eliminated from  $O_1.Type$ . Further, by including Mixture within the negative sign in  $O_1.Type$ , the definition becomes unnecessarily restrictive, as it prevents partial branching, where an output flow is also a mixture, as shown in Figure 10.5. Here, the input flow is a mixture of three components, while one of the output flows is a mixture of two components. As this situation is deemed common in mechanical systems, the new definition is required allow partial branching. This can be achieved by lifting the restriction from  $O_1.Type$ .



**Figure 10.5: Partial branching of mixtures not allowed by the definition of Branch**

In addition, the present definition allows the model in Figure 10.6, which satisfies the definition but violates the laws of conservation, as it converts an energy flow into a material. Thus, restrictions must be imposed to ensure that energy is only branched into other forms of energy, while materials are branched into other materials.

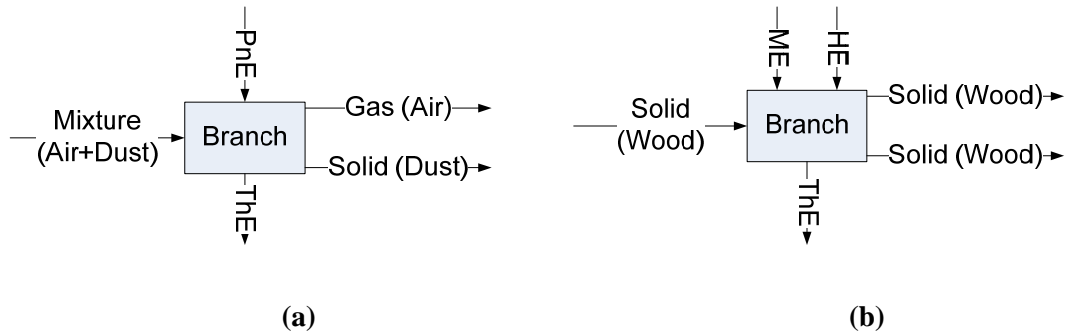


**Figure 10.6: Functions conflicting with natural laws are supported by the definition**

At present, signal flows are deliberately excluded from the definitions, as they need additional constructs of flow representation. For example, signals are neither conservable, nor independent entities, such as material or energy. They are carried by a material or energy carrier flow [183], and are interpreted by the recipient by sensing parameters of those carrier flows, such as the voltage of an electrical signal or the frequency of a laser. Thus, the modeling of signals requires modeling the carrier-carried relation, which is saved for the future. The definitions will be extended to include signals as and when this signal formalism is developed.

### Identify Enabler Flows in “Branch”

The existing definition cannot represent the exchange of energy between the system and its surroundings that may be required to execute the branching operation. While the branching of energies shown in Figure 10.4 does not require any enabling energy, typically the branching of material mixtures into its components requires energy exchange. In the case of passive branching, as with the dust filter in a vacuum cleaner (Figure 10.7a), this energy comes from the incoming air flow (Pneumatic Energy), as a result of which the outgoing air contains less energy than the incoming air and the filter heats up to dissipates the difference (Thermal energy). In the case of active branching, as with splitting a piece of wood with a table saw (Figure 10.7b), energy must be input to the system through the saw blade (Mechanical Energy) and the human hand (Human Energy), or through the electrical cord (Electrical Energy), if the whole table saw is considered as a system. Some energy also exits the system as heat (Thermal Energy) in this case. Both of these cases illustrate the natural law that some energy is needed to break the bonds between the components of a material flow, which is the essence of branching. Thus, energy flows that “enable” the branching operation must be included when the main flow is a mixture of materials. For energy main flows, the enablers should be optional, thus supporting the unlikely case where enabler energies are required to branch energies.



**Figure 10.7: Energy exchange between the system and surrounding cannot be modeled**

Topological Rules for the Enabler Flow Stream in “Branch”

Count check: The definition must allow multiple input enabler flows, as some branching operations may need that. For the sake of conservation, once a set of energy flows are introduced to the system, they must also exit the system. Thus, a set of output flows that represent the transformed form of the incoming enabling energies should be included in the definition. In addition, checks should be provided to ensure that the output flows could be modeled only when the input enabler flows were modeled.

Type check: However, at present, these enabling flows are only envisioned as energy flows, although material or signal enables may be included in future.

Conservation Rules for “Branch”

At present, mixing between the main and the enabling flow streams is not perceived as a modeling requirement. Thus, the mass and energy of all flows in *In\_List*

must be conserved in *Out\_List* and all input enabler flows must also be conserved by their output forms.

#### Formal Class Definition of “Branch”

The textual definition of Branch [26] is formalized in Figure 10.8. This definition, presented in first order logic statements as pseudo code of an object oriented class, is meant to capture the intent of the verb’s textual definition, while adding the necessary constructs to resolve the inconsistencies and to ensure adherence to the conservation laws, and ultimately expresses it in a formal, syntactic form so that the computer can reason on the definition at the syntactic level in an algorithmic manner, rather than relying on human semantic interpretation. The three sections in this definition, marked by the commented lines, indicate results of the four checks in the protocol. *Enabler\_In\_List* and *Enabler\_Out\_List* are initialized as optional arguments and default to the empty list, as the function may not always need enablers. The symbol  $\mathbb{Z}$  represents the integer set. The statements here are simultaneously enforced and should be read as connected by logical AND conditions ( $\cap$ ). These symbols are omitted for the sake of readability.

```

Class : Branch{In_List, Out_List
    Enabler_In_List = {}, Enabler_Out_List = {}
{
/* Topological rules: Main stream of flows*/
    In_List = {I_1} // Accept only one input flow to be branched per instance
    Out_List = {O_1, O_2, ..., O_m} // Have provisions for producing up to m flows
    m > 1, m ∈ ℤ // Must produce at least two output flows - branching action
    I_1.Type ⊆ M ∪ E // The input flow can be a material or an energy flow
    I_1.Type ⊆ M → ∀i ≤ m, O_i.Type ⊆ M // If input is M, all output must be M - No conversion from
    // M to E is allowed
    I_1.Type ⊆ E → ∀i ≤ m, O_i.Type ⊆ E // If input is E, all output must be forms of E - no conversion
    // from E to M is allowed

/* Topological rules: Enabler energy flows */
    Enabler_In_List = {J_1, J_2, ..., J_p} // Multiple enabler flows are allowed to enable branching
    Enabler_Out_List = {K_1, K_2, ..., K_q} // The enabler flows can be output as multiple flows
    p ≥ 0, p ∈ ℤ // Optionally, an instance of Branch may not need any enabler
    q ≥ 0, q ∈ ℤ // Same optional flexibility for other output enabler flows
    p = 0 → q = 0 // If no enabler input, there must be no enabler output
    // for the sake of conservation
    p > 0 → q > 0 // If there is an enabler input, at least one enabler output must
    ∀i ≤ p > 0, J_i.Type ⊆ E // In the present version, only E flows can be enablers
    ∀i ≤ q > 0, K_i.Type ⊆ E // Consequently, only E flows are allowed as enabler output

/* Conservation Rules */
    I_1.Mass = ∑_{i=1}^m O_i.Mass // Main flow mass conservation
    I_1.Energy = ∑_{i=1}^m O_i.Energy // Main flow energy conservation
    ∑_{i=1}^p J_i.Energy = ∑_{i=1}^q K_i.Energy // Enabler energy conservation - separately from main flows
    // No mass conservation for enablers required, since only E
    // flows can be enablers
}

```

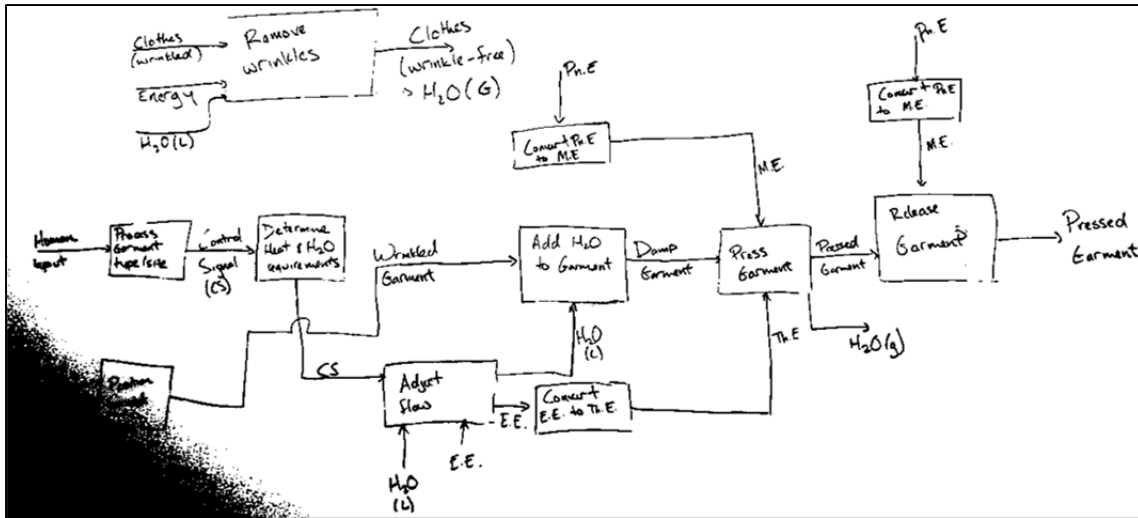
**Figure 10.8: Formalized definition of Branch consistent with the conservation laws**

Once a significant set of notional verbs are formalized through this protocol, they will be encoded in ConMod and used to perform conservation and irreversibility-based reasoning. This work is reserved for the future. While this direction is in progress, another necessary direction to explore is the practical usability of design tools supported by the proposed representation to human designers. This direction of future work is discussed next.

### **10.5.2 Ongoing Work: Examination of Designer-Level Usability**

While this dissertation establishes internal consistency and external validity of the representation, it does not go into examining the requirements for software implementation of the representation. For example, the ConMod software tool developed within this dissertation is meant primarily for demonstration of reasoning ability, rather than for use in actual design projects by end-users. Although ConMod, in its present form, could be used in design projects, it is understood that the practical usefulness of the representation intimately depends on the usability and software design aspect of the software implementation. The ultimate success of this research will not be realized until it is implemented in tools that designers can use. A research direction is already underway to find the requirements for these tools, where designers are studied through protocol analyses [184] to reveal patterns of their interaction with a function structure model. Preliminary studies in this direction have helped establish rich protocols that will be next applied to larger participant pools. Figure 10.9 shows a sample model produced by a participant in this experiment.





**Figure 10.9: Sample results of pilot protocol studies on designer-model interaction**

In this case, the participant is given the following novel design problem and asked to construct function structure models to explore functional architecture for the design. Two participants, P1 and P2, were used in this reported pilot study.

Design Problem for Protocol Study

*“Design an automatic clothes-ironing machine for use in hotels. The purpose of the device is to press wrinkled clothes as obtained from clothes dryers and fold them suitably for the garment type. You are free to choose the degree of automation. At this stage of the project, there is no restriction on the types and quantity of resources consumed or emitted. However, an estimated 5 minutes per garment is desirable”.*

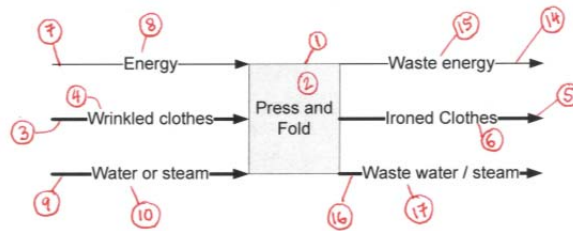
The resulting models are put through a formalism of encoding, completed in three steps: activity encoding, element encoding, and topology encoding. The first step is a time study of the sequence of different activities through the modeling session. The

second is a cataloging of the model elements created through the sessions, while the last is an accounting of the modeling process, which accounts for how each element was connected to the surrounding elements at creation-time. Collectively, these three steps encode the entire modeling process, such that both the model and the modeling process can be re-enacted purely from the encoded information.

STEP 1: Activity Encoding

In activity encoding, the timestamp of starting an activity and the elements produced in that activity are encoded in a spreadsheet such as Figure 10.10a. The elements are given unique IDs that are tracked throughout the experiment, as shown in Figure 10.10b for one participant, P1.

TmStmp	Act	Element IDs					
0:30	PS						
1:21	A	1	2	3	4	5	6
1:48							
2:10	PS						
2:18							



(a) Activity encoding sheet

(b) Assigning IDs to elements for P1

**Figure 10.10: Activity Encoding for Participant P1**

STEP 2: Element Encoding

In this step, the type of each element is encoded against its unique ID, thus producing a table such as Figure 10.11 (first two columns). Here B = block, BT = block text, E = edge, ET = edge text.

### STEP 3: Topology Encoding

In this step, the connectedness between the elements is captured by recording the IDs of the elements at the “head” and “tail” of a given element, at the instant when the given element is drawn by the participant. For edges, the head and tail block IDs are recorded, for blocks, the input and output edge IDs are recorded, and for texts, only the parent block or edge IDs are recorded as tail, while the head is left blank. These recordings are shown in the last two columns of Figure 10.11.

Elem ID	Elem Typ	Topology	
1	B	0	0
2	BT	1	
3	E	0	1
4	ET	3	
5	E	2	0
6	ET	5	
7	E	0	1

**Figure 10.11: Element encoding for participant P1**

Several critical observations are made from the encoded data. For example, 46 of the 48 flows in the resulting model made by P1, the designer added the flow name (ET) immediately after adding the flow (E). Yet, for thirteen of the seventeen functions, the function name was added only after adding their attached flows, or at least after a pregnant pause. This trend was also strongly visible in participant P2’s activity sheet. Thus, the flow type or name was known to the designers as soon as a flow was conceived, indicating that these designers conceived the device in terms of the flows it would process, rather than in terms of functions it would perform. The function names

were later retro-fitted to describe the resulting transformative actions indicated by the flows.

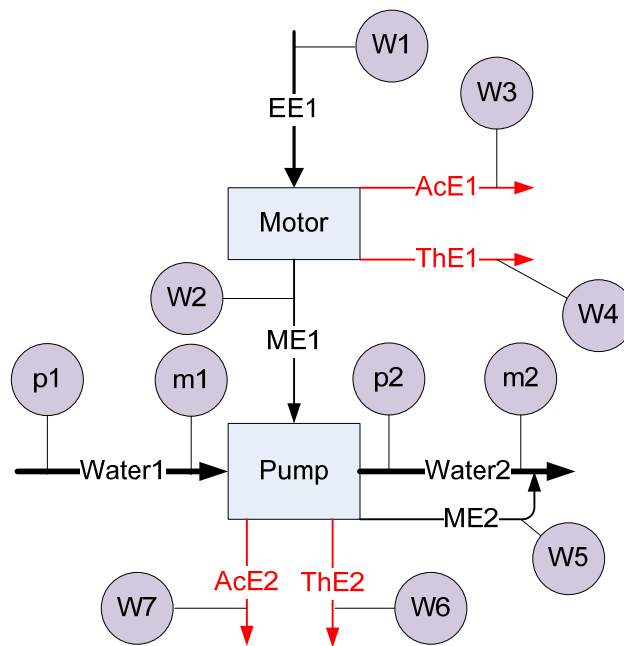
P1's overall approach was nucleation, as he started the decomposed model with a few sub-functions on each end of the board, indicating the clearly identified sub-actions involved in ironing, and eventually finished the model by connecting those functions and others through edges in the middle of the board. P2, however, followed a generally forward chaining approach. The first function drawn was a subfunction on the left end of the board and the last function was the final action that produced folded pressed clothes.

However, when the modeling actions are observed at a finer time-resolution, both designers seem to use nucleation and forward chaining when adding functions, and forward and backward chaining when adding flows. For example, P2 added six functions by nucleation, seven functions by forward chaining on the head of an existing flow, but only one function by backward chaining, on the tail of a flow. However, for the flows, P2 added sixteen and eleven flows by forward and backward chaining, and only two through nucleation, which were later appended with functions through forward chaining.

Several other observations are made about the rate of modeling, patterns of model editing and deleting, and the use of the black box model in developing the decomposed model. These details will be used to further refine the design of the protocol and execute the study at larger scales. The next section discusses the evolving work on causal reasoning supported in the future.

### 10.5.3 Future Work: Causal Reasoning Extension

Causal reasoning is a potential candidate for a future extension of this research. Figure 10.12 shows the function structure of an electric motor and pump assembly, where the intended and residual flows are identified. Causal reasoning at a quantitative level would require capturing numeric values of flow and function parameters, as illustrated during the quantitative extension demonstration of Chapter 7. All energy types have a power attribute ( $W$ ), all liquid flows have a pressure attribute ( $p$ ), and all material flows have mass ( $m$ ).



**Figure 10.12: Function structure of an electric motor and pump assembly**

An example reasoning question asked on this model could be: “Determine the effect on the design if the incoming water flow (Water1) is stopped”. In order to answer this question, the reasoner must use the knowledge of causal relations between functions

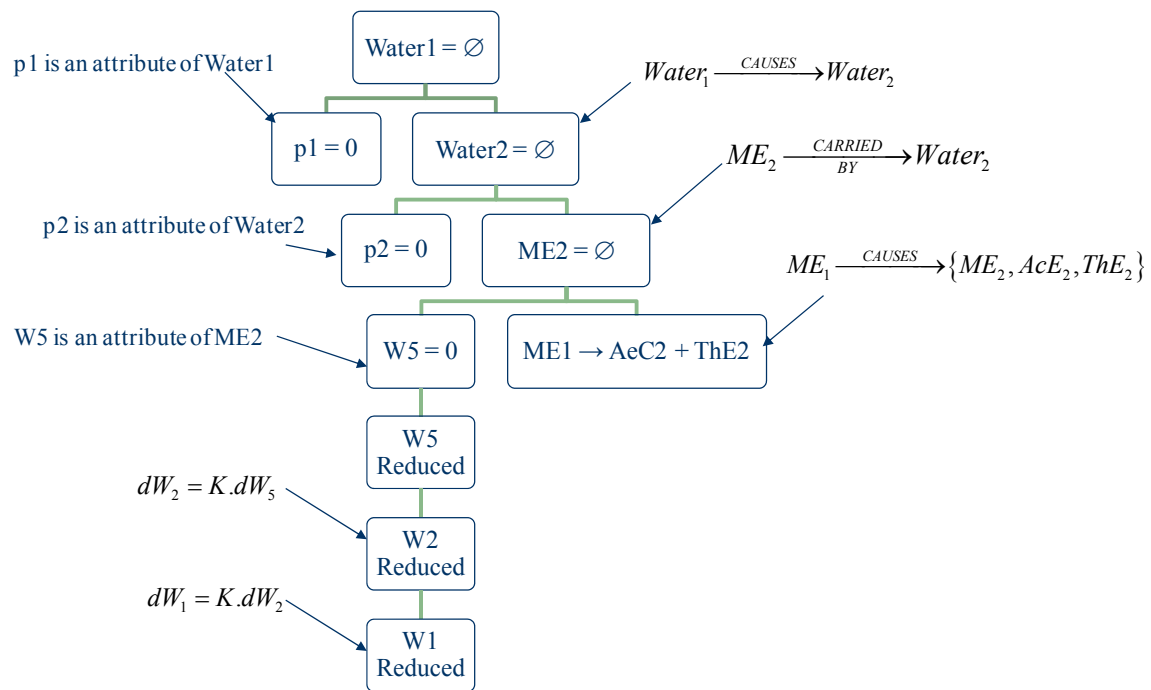
and flows. For example, it must know that  $EE_1$  causes  $ME_1$ ,  $AcE_1$ , and  $ThE_1$ —a knowledge element that can be inferred directly from the conservation relation already captured in the representation. Similarly, it can be automatically reasoned that all outgoing energy flows from the pump are caused by  $ME_1$ . This direct usability of the conservation relation in some cases to derive the causal relations makes causal reasoning a strong candidate for future extension of this research.

$$\begin{aligned}
 EE_1 &\xrightarrow{CAUSES} \{ME_1, AcE_1, ThE_1\} \\
 dW_1 &= K.dW_2 \\
 ME_1 &\xrightarrow{CAUSES} \{ME_2, AcE_2, ThE_2\} \\
 dW_2 &= K.dW_5 \\
 Water_1 &\xrightarrow{CAUSES} Water_2
 \end{aligned}$$

**Figure 10.13: Causal relationships to be captured in the extended representation**

However, causal relation is not always derivable from conservation. For example, if Water 1 is stopped because there is no water at the suction end of the pipe (sump), then entire inside of the pump will run dry, causing the impeller to rub freely without doing work to pressurize the water, and consequently the motor would consume less power. In a different case, if Water 1 stops because the suction pipe is choked, the inside of the pump and the water lines are still filled with water (primed), causing the impeller to spin inside still water, and consequently the motor may consume more power. Thus, it may not be enough to describe the root cause as “Water 1 stopped” but the cause for such stoppage must be mentioned and used in reasoning. The future research directions will include formalizing these cause and effect statements, their logical relations, and the algorithms to computationally reason in terms of effect propagation. Figure 10.14

describes a possible causal reasoning tree for such a derivational algorithm, where Water 1 stops due to the sump running dry. The blocks indicate inferences drawn by the reasoner, the tree branches are the sequence of causal inference propagation, whereas the externally drawn arrows indicate the information elements within the model that the reasoner would use to draw the inferences.



**Figure 10.14: Causal Reasoning Tree**

In this tree, the root node describes that the flow  $Water_1$  has stopped. Since  $Water_2$  is modeled as an effect of  $Water_1$ , it can be reasoned that  $Water_2$  will stop, causing  $ME_2$  to cease. Next, it will be possible to reason that all the energy entering the pump through the shaft ( $ME_1$ ) will be spent in producing only heat ( $ThE_2$ ) and sound ( $AeC_2$ ). Additionally, since  $ME_2$  has ceased,  $W_5 = 0$ . Based on the quantitative relations, it can be determined that  $W_2$  and by effect  $W_1$  will also reduce. Thus, it will be possible

to reason, from the facts presented explicitly in the model, that if the incoming water flow is cut off, the outgoing water flow will stop, all the energy entering the pump will be wasted as heat and sound, the pump will consume less power than before, and the motor will draw less power than before.

#### **10.5.4 Future Work: Additional Directions to Explore**

A major possibility identified in this dissertation is to extend the representation with quantitative attributes of functions and flows, which would potentially support more enhanced reasoning. In this context, the need for integrating the representation with a variational solving system is discussed in Chapter 7. With a variational solver, the design rules of a specific domain could be captured as non-directional rule statements, unlike parametric statements, and the tool could be used to solve for any of the unknown parameters, as long as enough numeric data for computing its relations are available in the model.

Another area identified but not addressed in this dissertation is the need to support resolution scaling and decomposition-based reasoning. Model decomposition and problem discovery is a major anticipated benefit of function-based modeling and thinking in design texts [1, 2]. Automating this area will help realize these benefits through computational reasoning.

While the dissertation proposes physics-based verbs and ongoing work on formalizing notional verbs are discussed in this chapter, an important area to simultaneously formalize is the set of nouns for flow modeling in formal function



structure models. This area is unaddressed in this research and must be developed in order to realize the complete benefits of this work.

As indicated in the hairdryer function model of Section 9.4.1, this representation is not capable of describing transient phenomena and only describes steady states of operation (use modes). Many mechanical engineering devices rely on transient phenomena for their operations and thus, a potentially useful extension of this work is in modeling those processes.

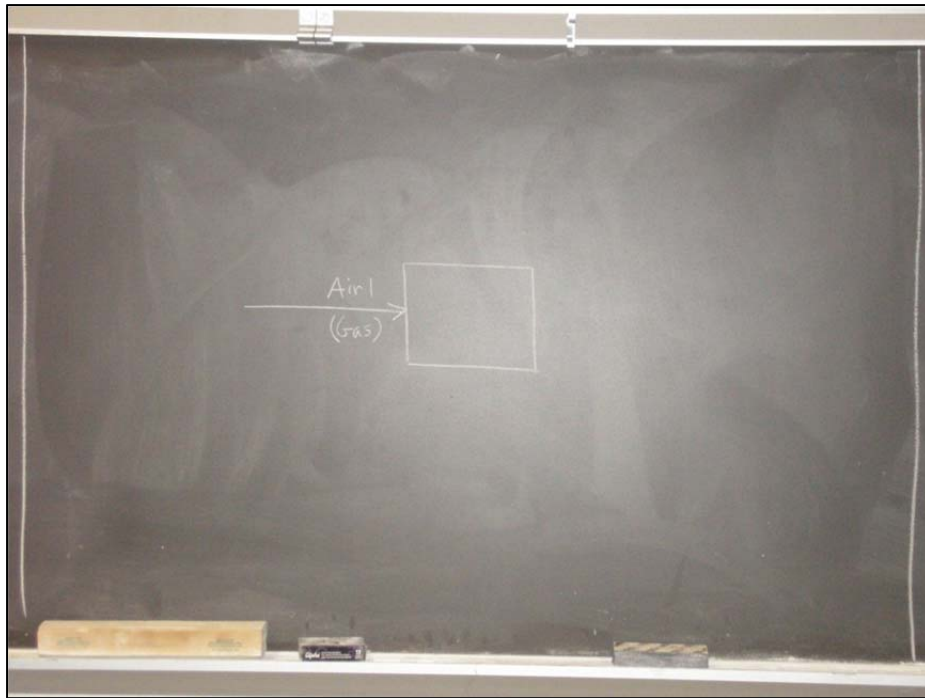
Finally, while this research is primarily focused on supporting analytical reasoning in early design, it is anticipated that it could be extended to support design synthesis. As shown with an example in Section 8.1, a software tool built on this representation could be connected to a database of solution principles and function structure models drawn in these tools could be used to search solution candidates from those databases. Several directions of contemporary research in function-based synthesis [6, 20-22, 98] identify criteria for ranking and selecting solution candidates in similar situations. It is anticipated that the current research and its future extensions can be integrated with these ongoing efforts to evolve a foundational representation and a distributed software framework, providing automation support to the entire conceptual design process through formal, rigorous, computational reasoning.

## APPENDICES

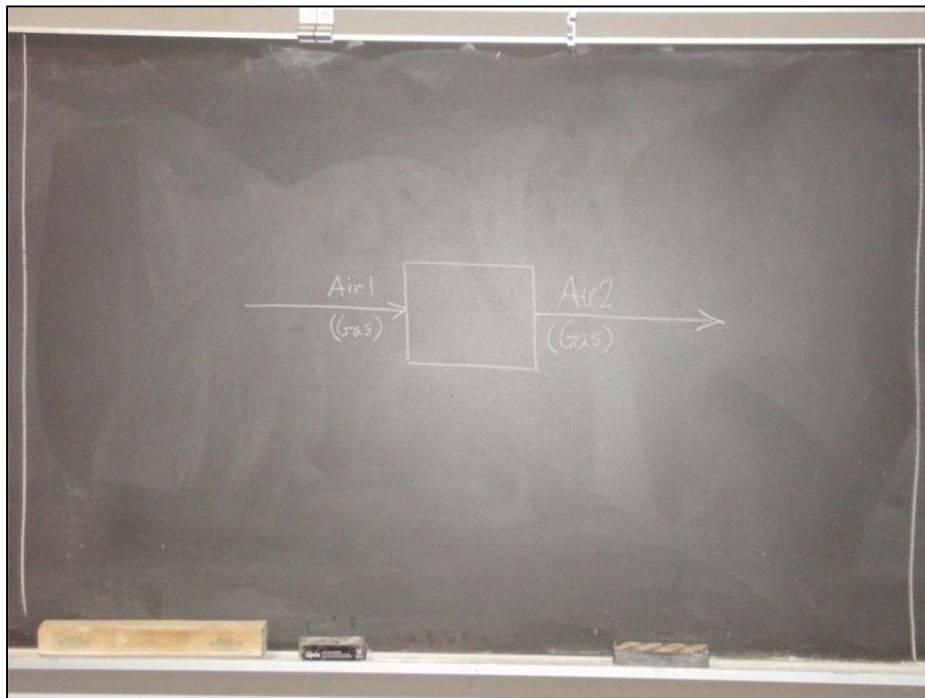
**Appendix A. Reasoning Discovery Experiment Steps Discussed in Section 4.1**



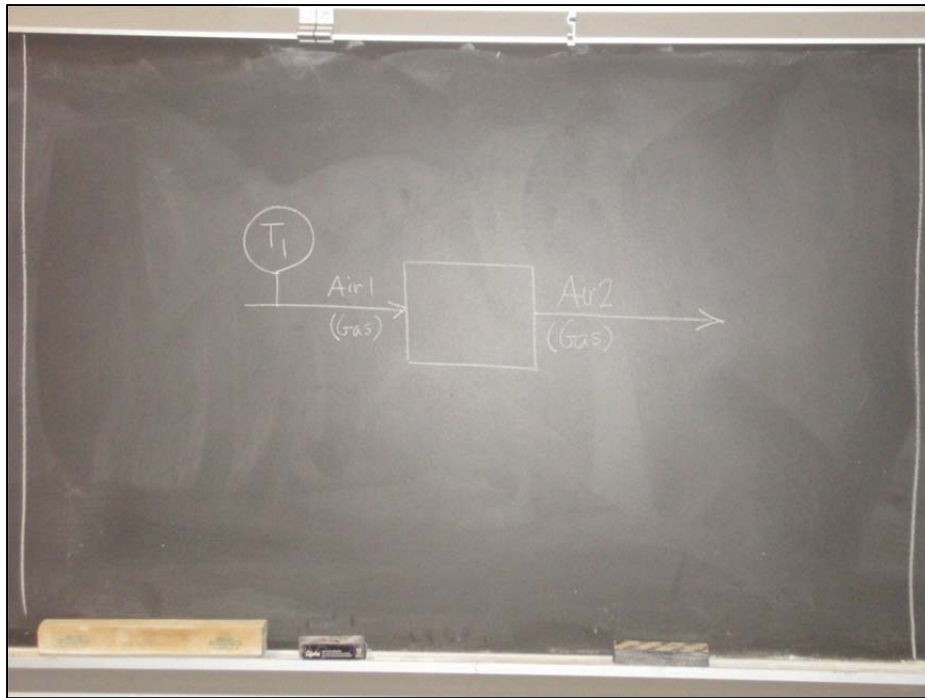
**Chalkboard Exercise Step 1**



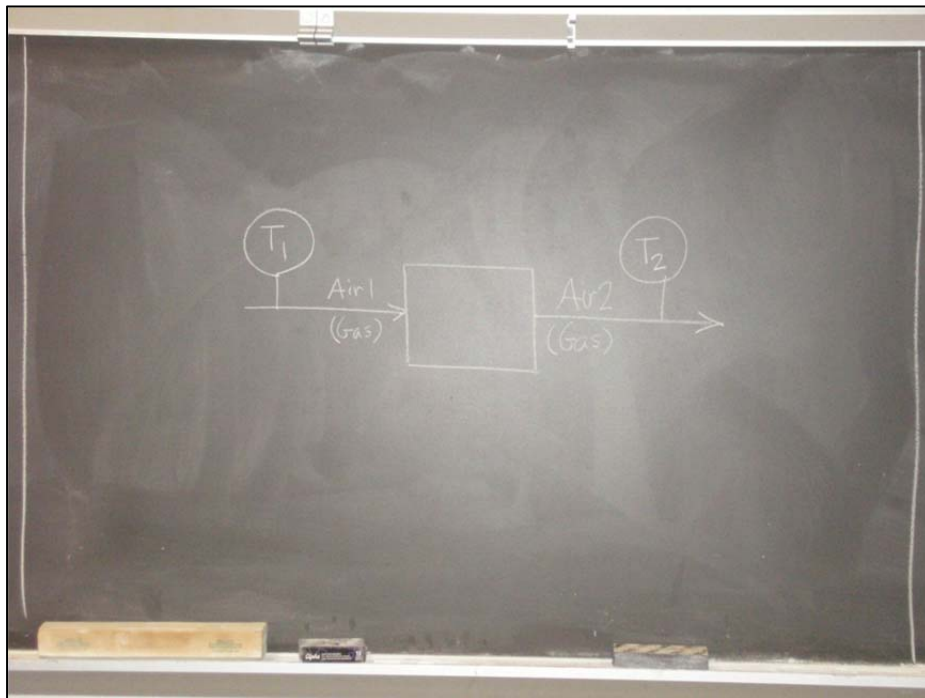
**Chalkboard Exercise Step 2**



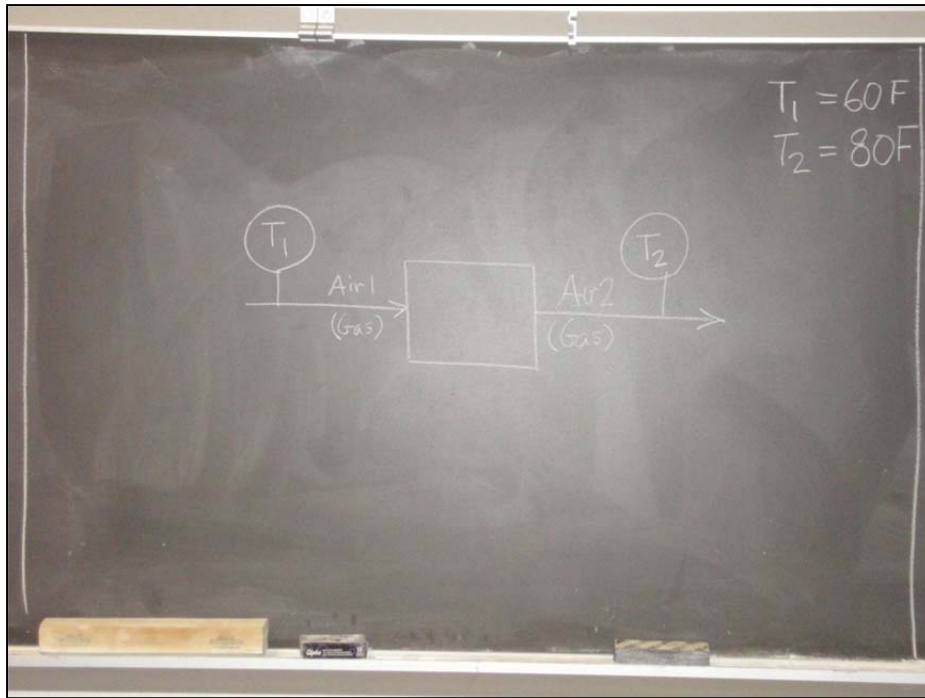
**Chalkboard Exercise Step 3**



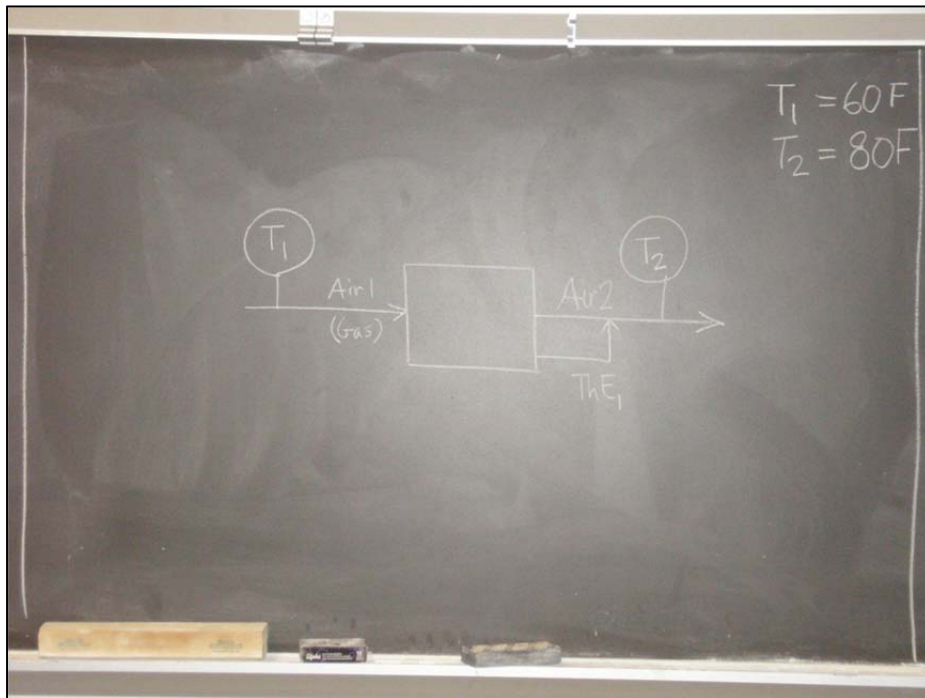
**Chalkboard Exercise Step 4**



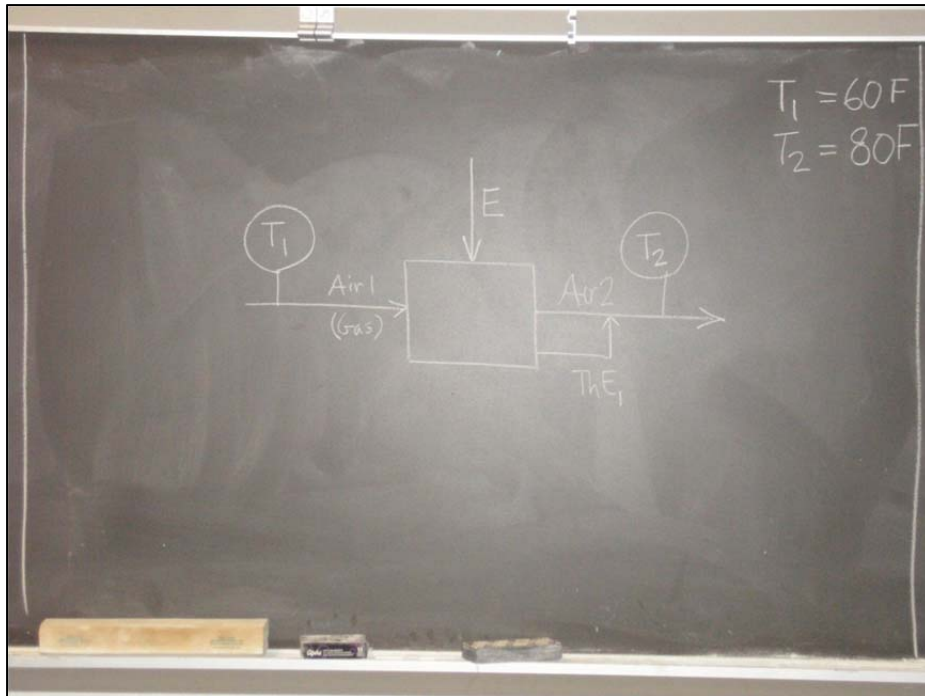
**Chalkboard Exercise Step 5**



**Chalkboard Exercise Step 6**



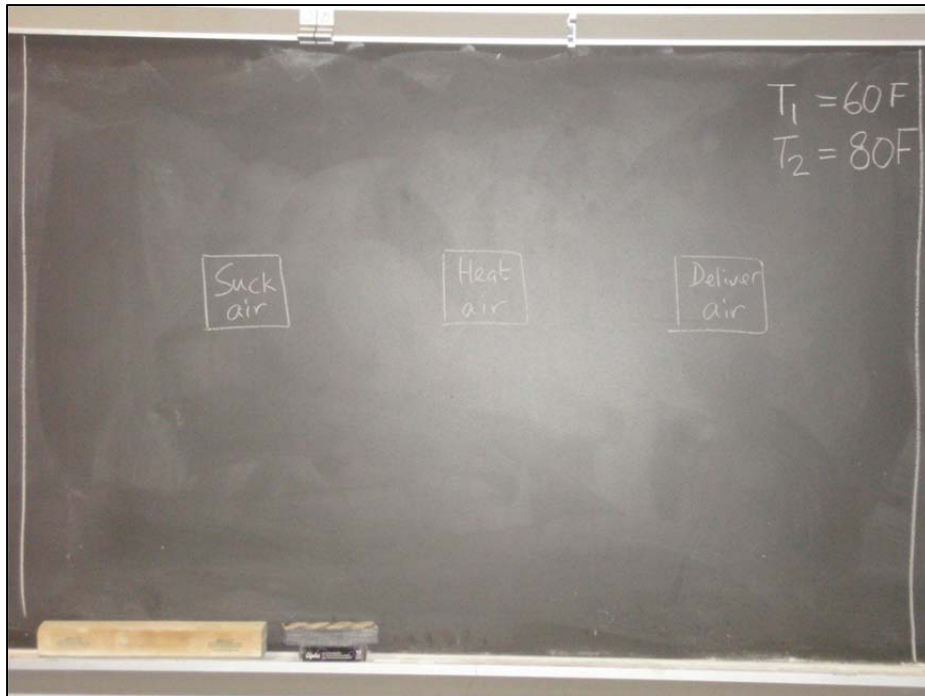
**Chalkboard Exercise Step 7**



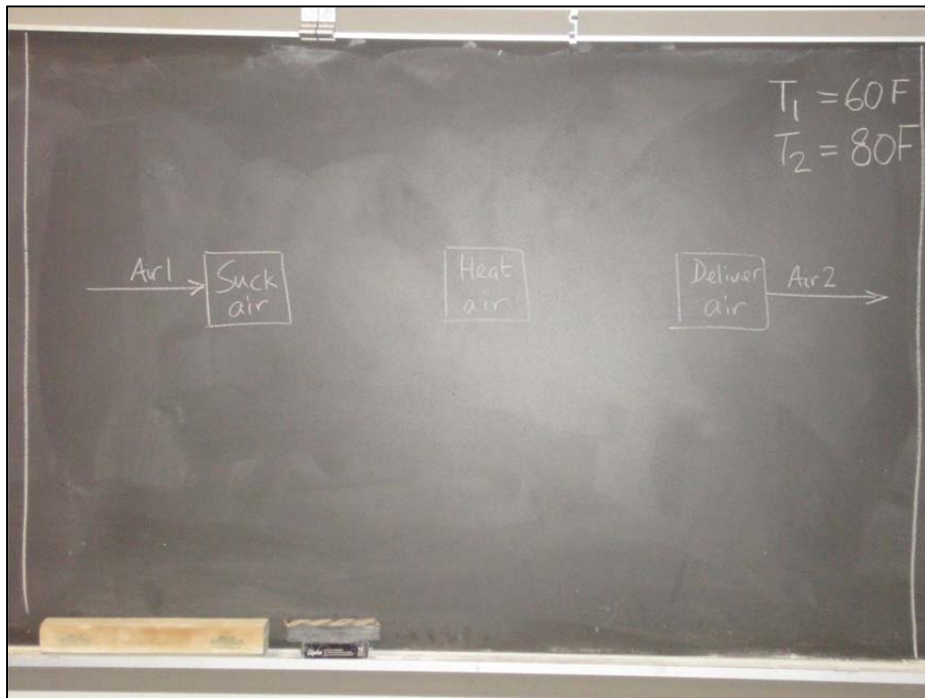
**Chalkboard Exercise Step 8**

A chalkboard with the following text written in the center: "Suck air", "Heat air", and "Deliver air". In the top right corner, the temperatures are listed as  $T_1 = 60F$  and  $T_2 = 80F$ .

**Chalkboard Exercise Step 9**

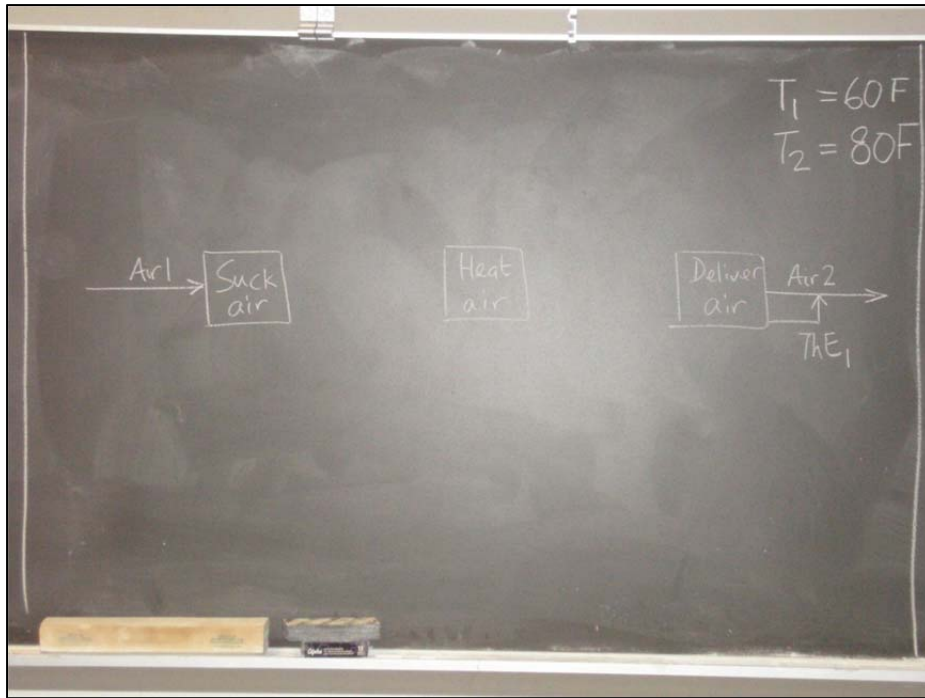


**Chalkboard Exercise Step 10**

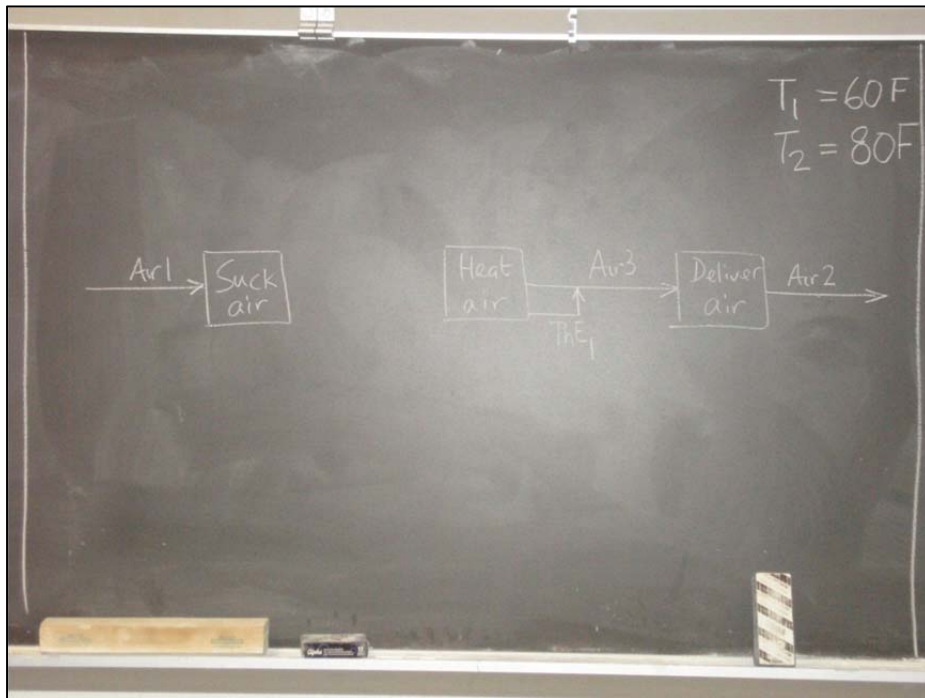


**Chalkboard Exercise Step 11**

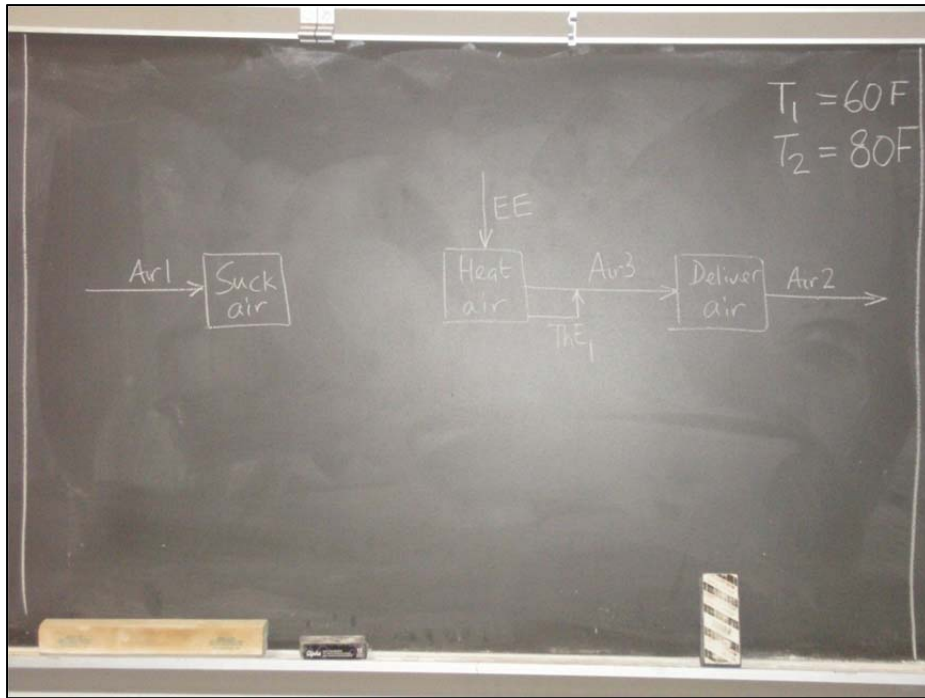




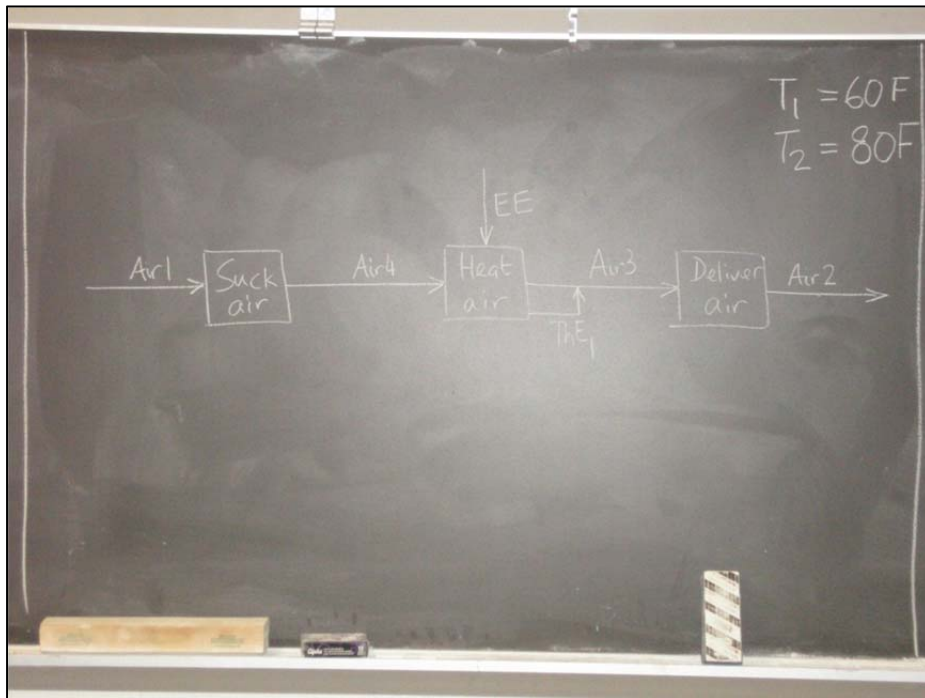
**Chalkboard Exercise Step 12**



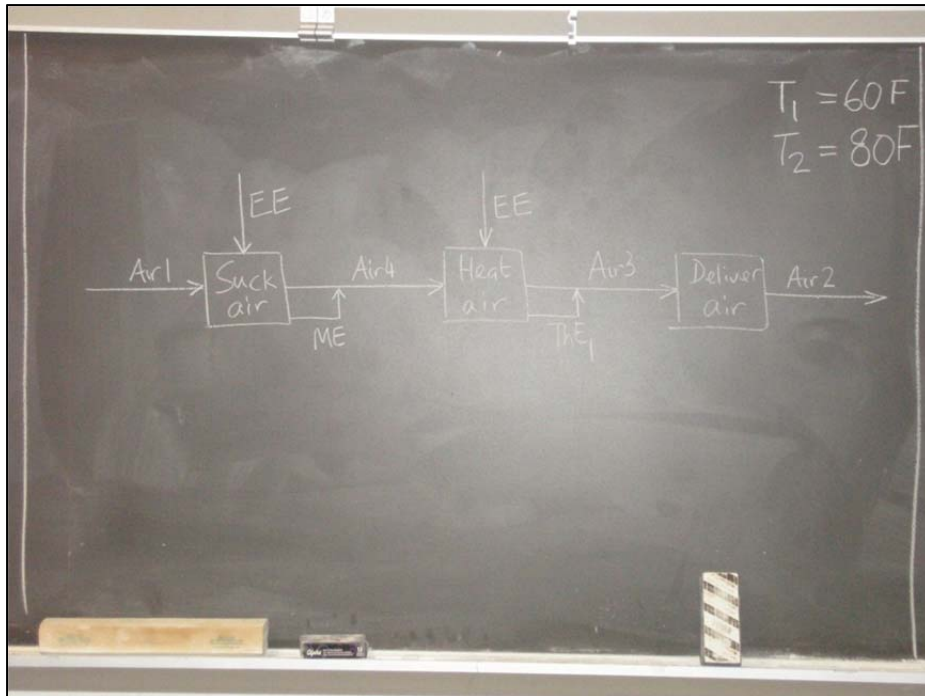
**Chalkboard Exercise Step 13**



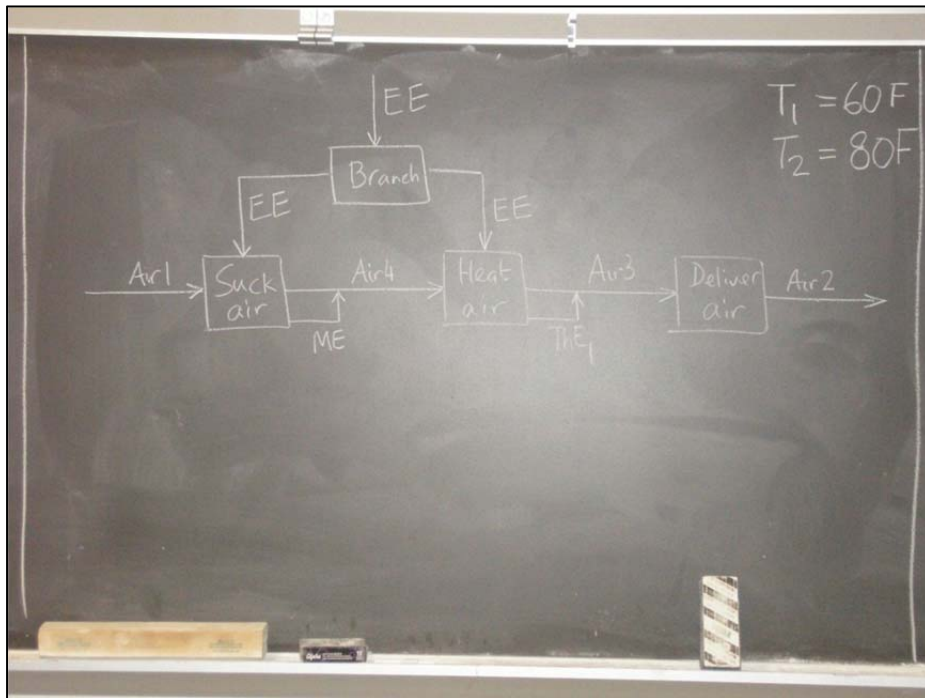
**Chalkboard Exercise Step 14**



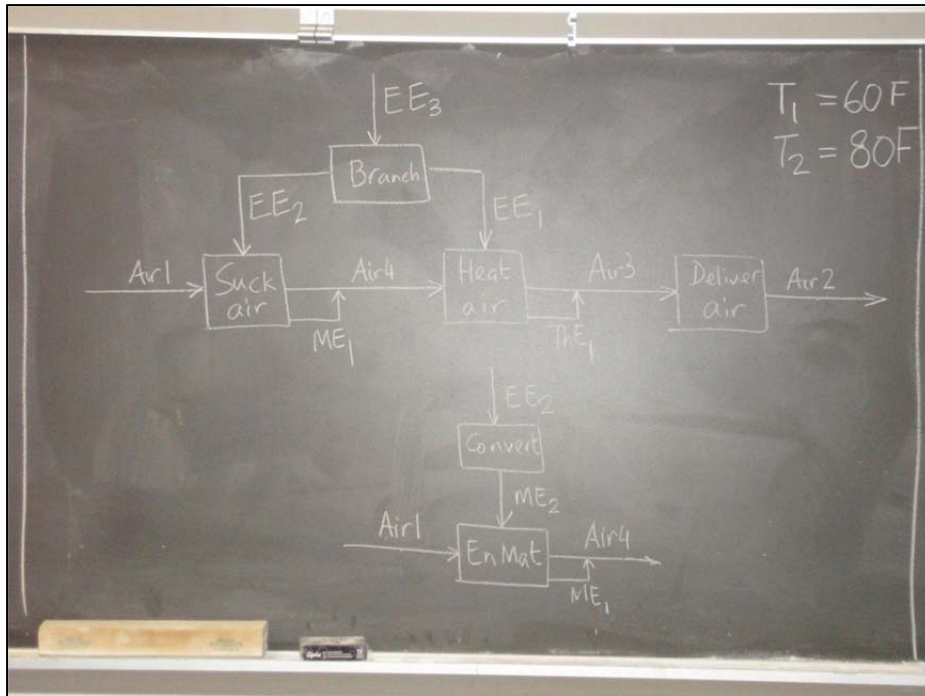
**Chalkboard Exercise Step 15**



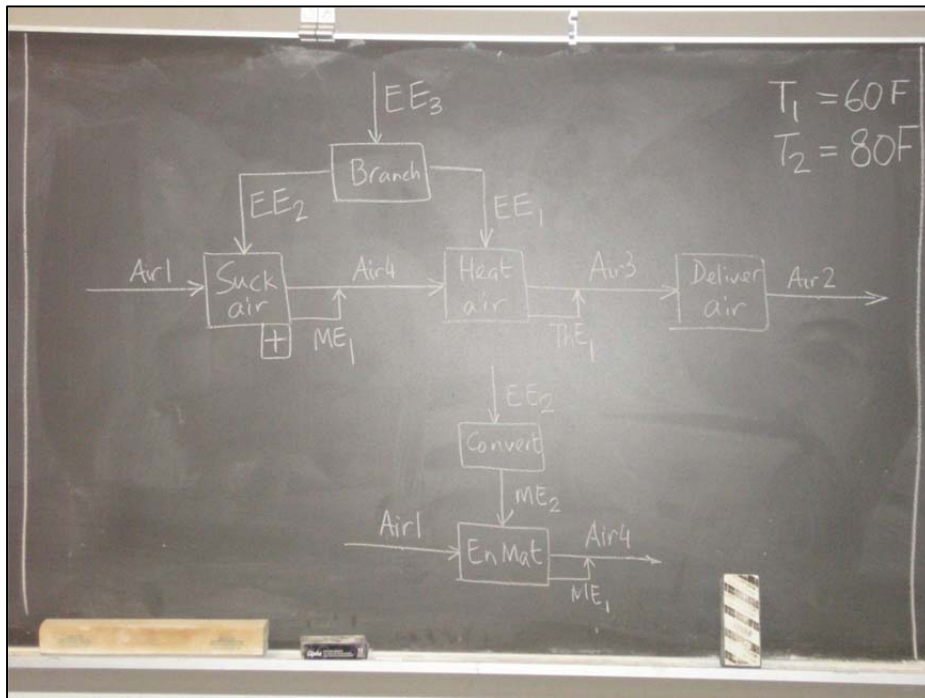
**Chalkboard Exercise Step 16**



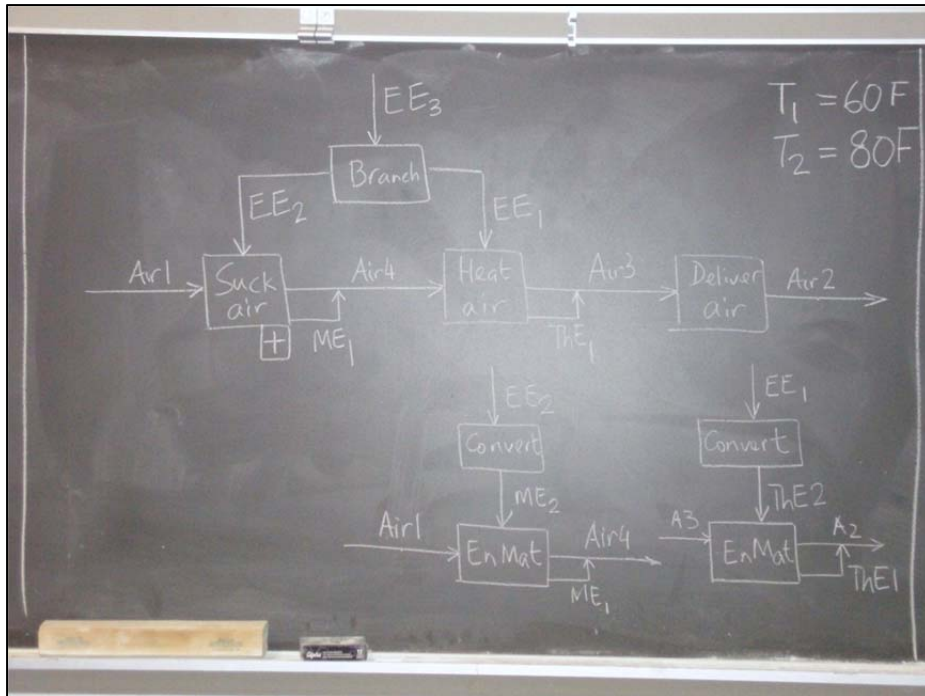
**Chalkboard Exercise Step 17**



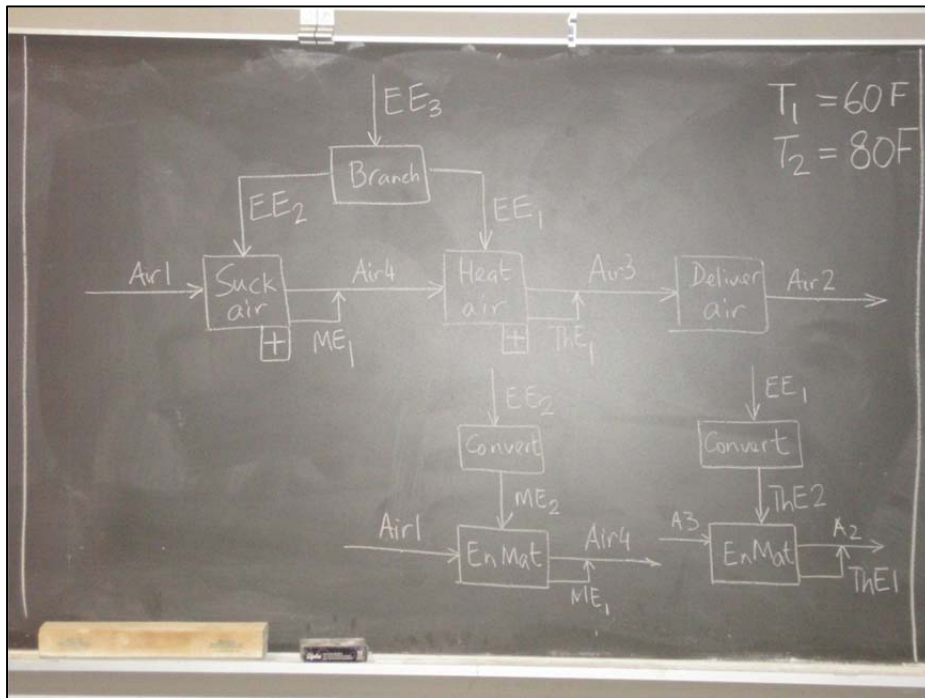
**Chalkboard Exercise Step 18**



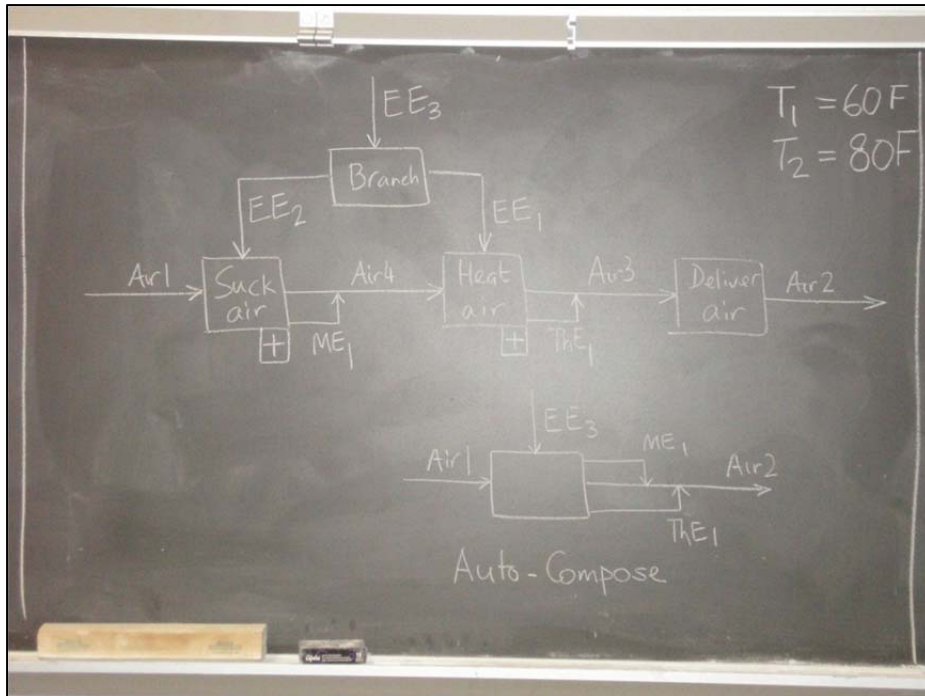
**Chalkboard Exercise Step 19**



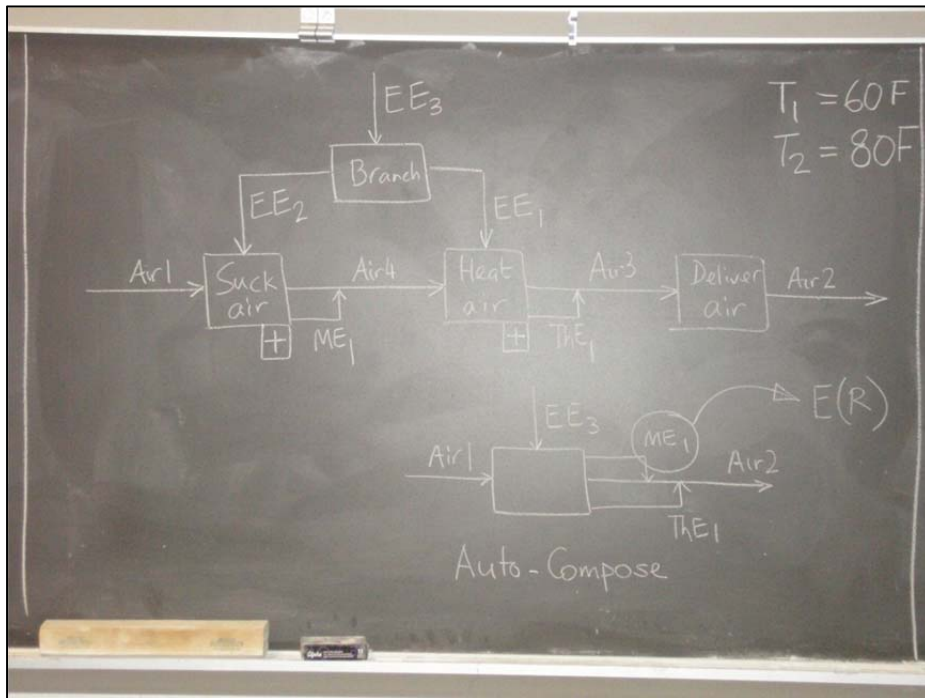
**Chalkboard Exercise Step 20**



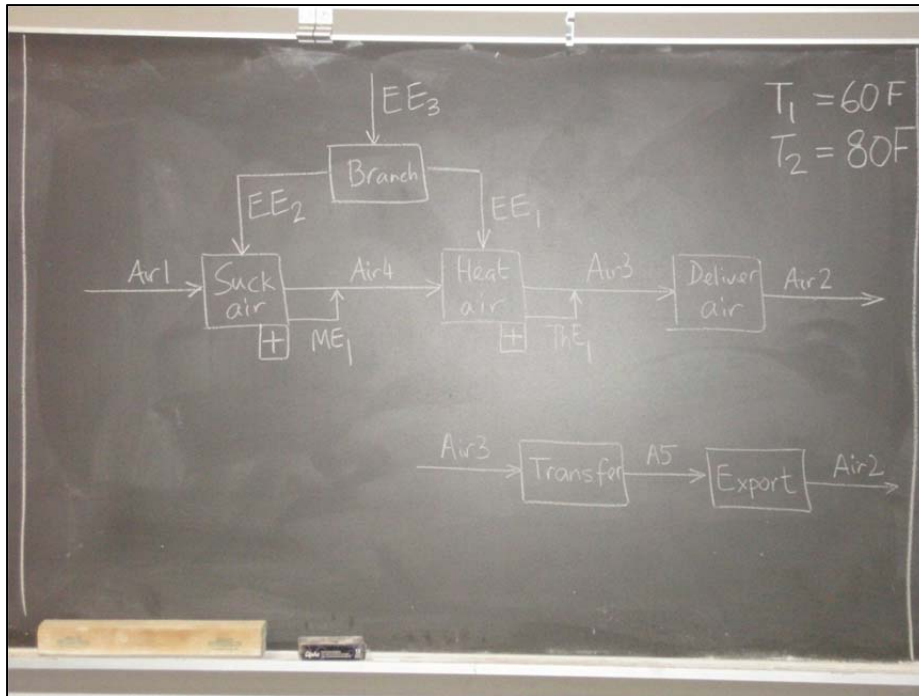
**Chalkboard Exercise Step 21**



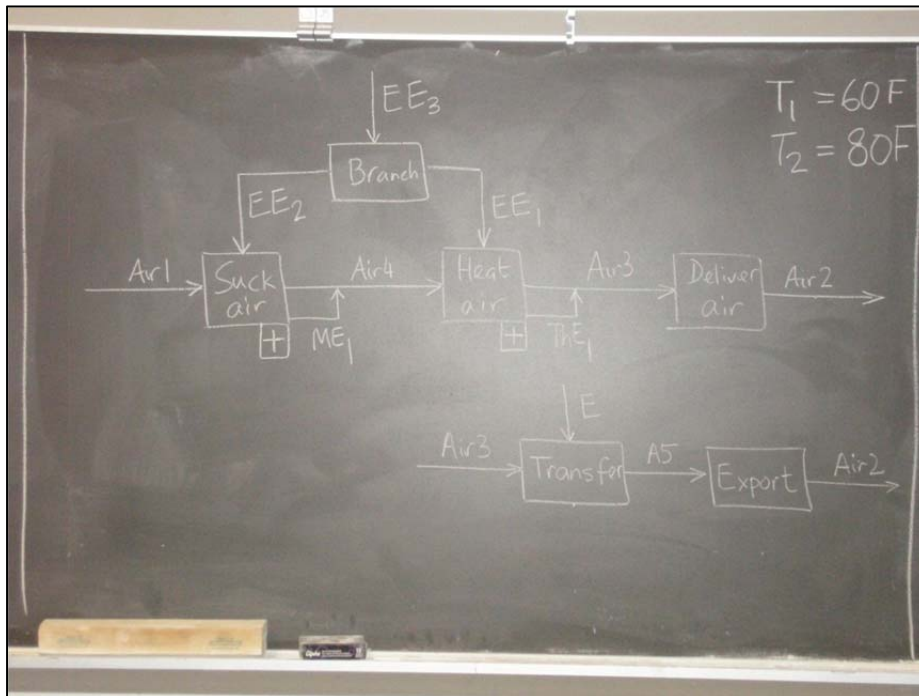
**Chalkboard Exercise Step 22**



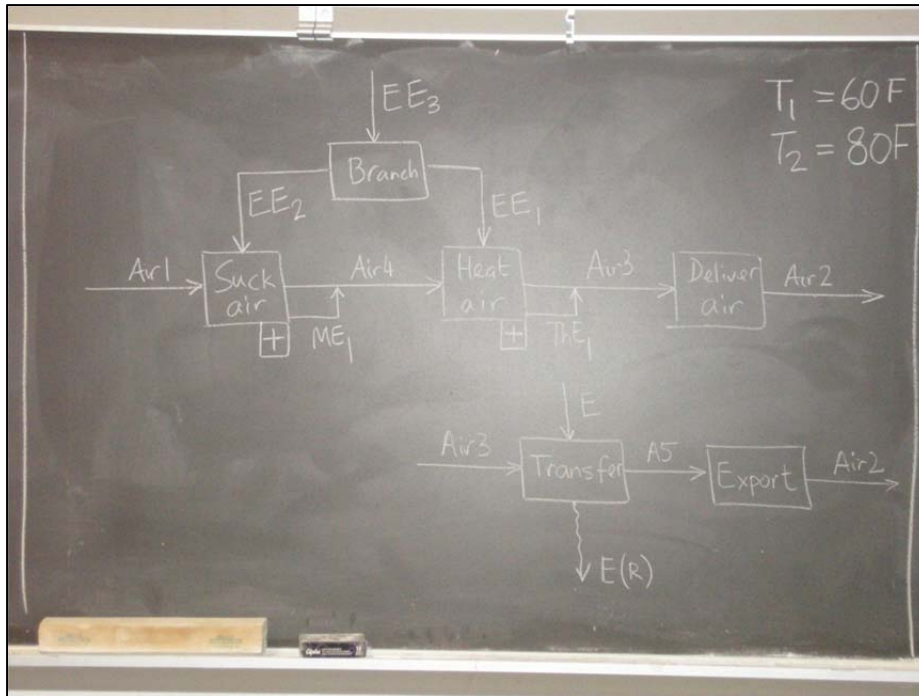
**Chalkboard Exercise Step 23**



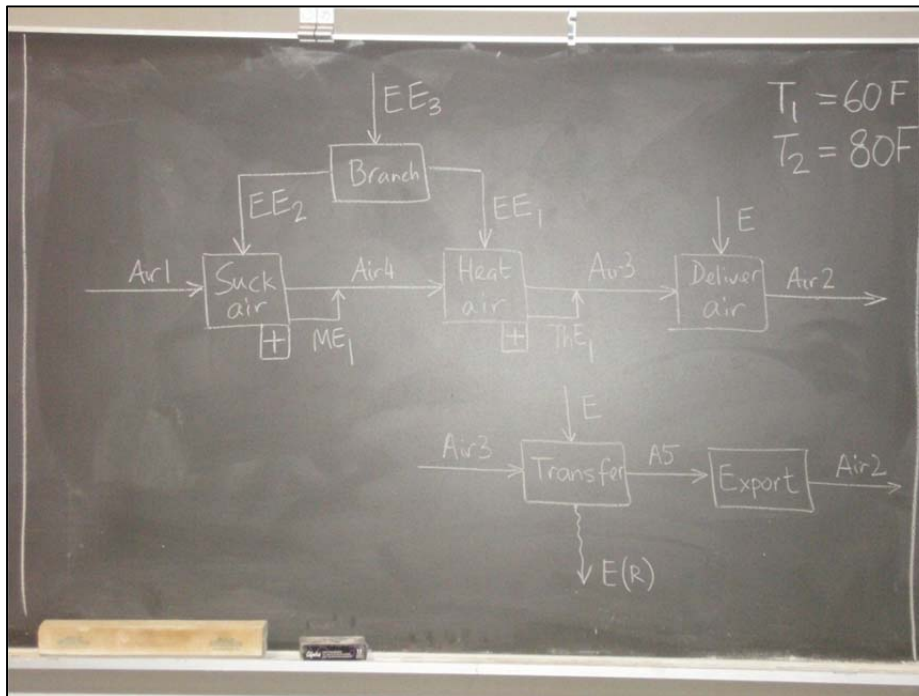
**Chalkboard Exercise Step 24**



**Chalkboard Exercise Step 25**

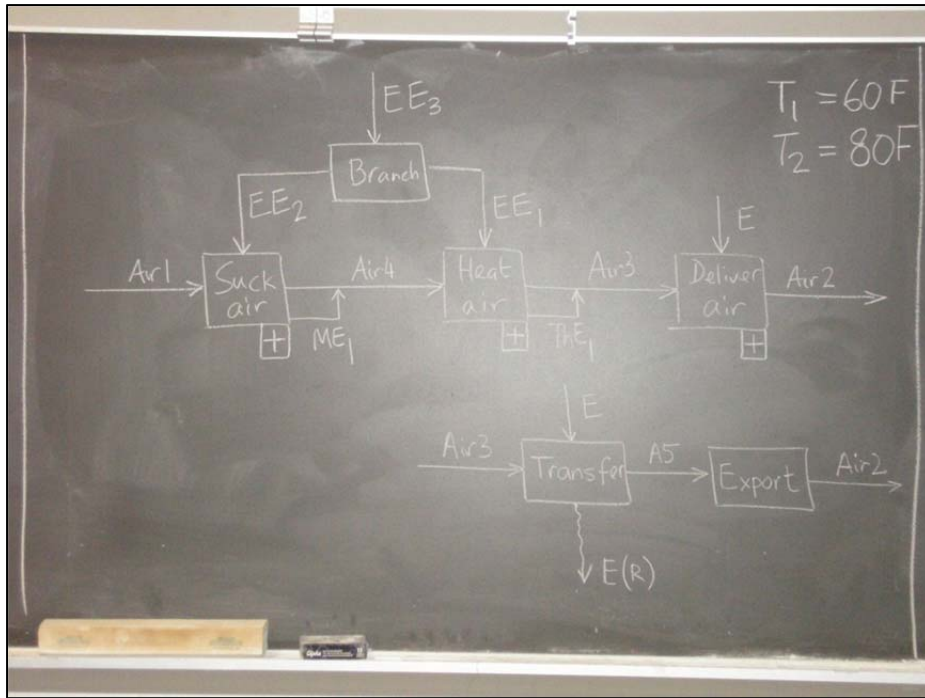


**Chalkboard Exercise Step 26**

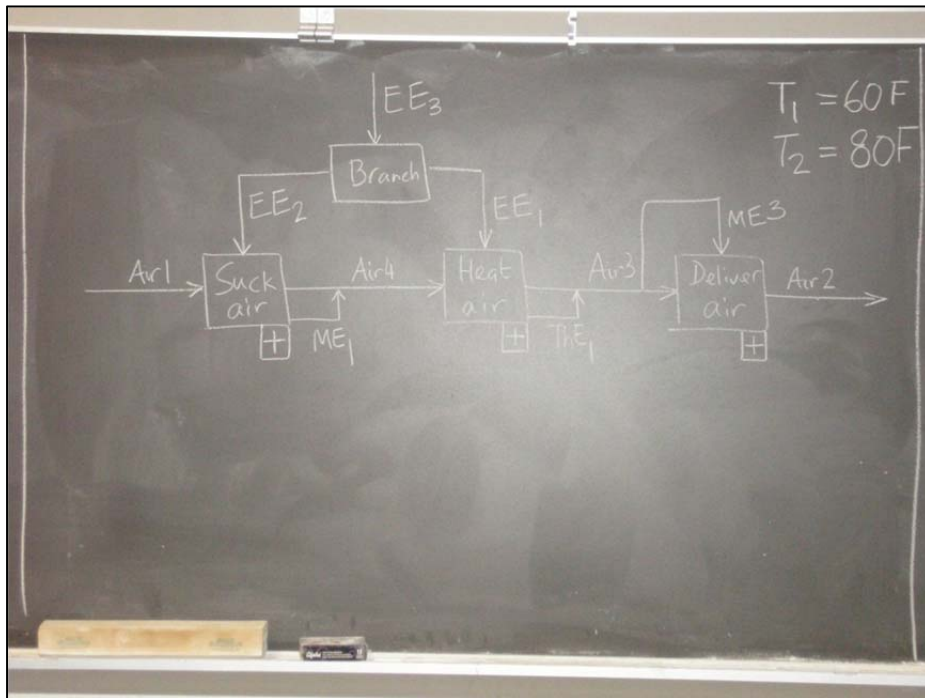


**Chalkboard Exercise Step 27**

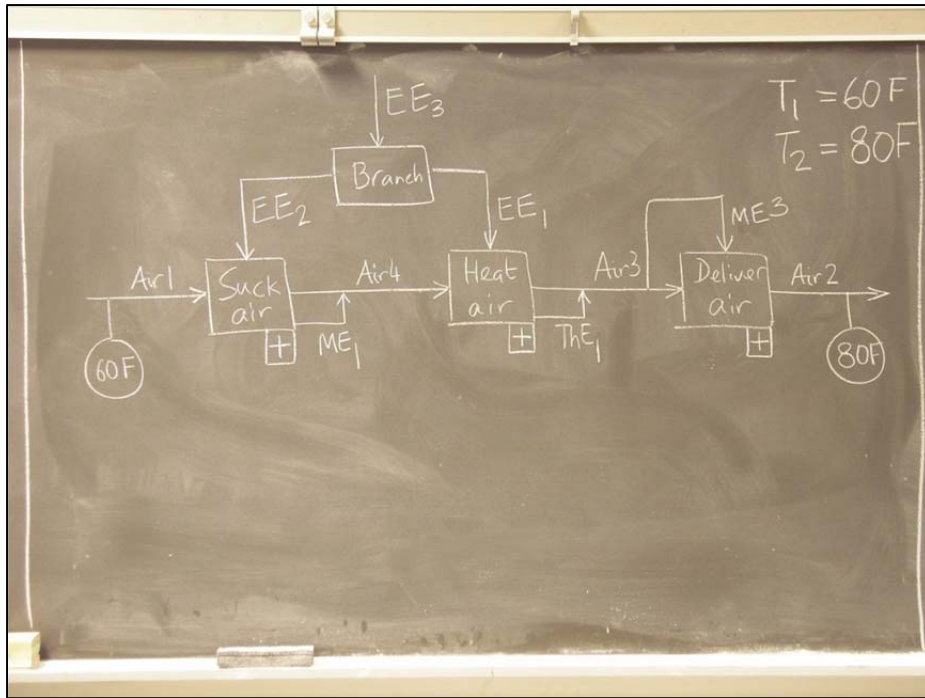




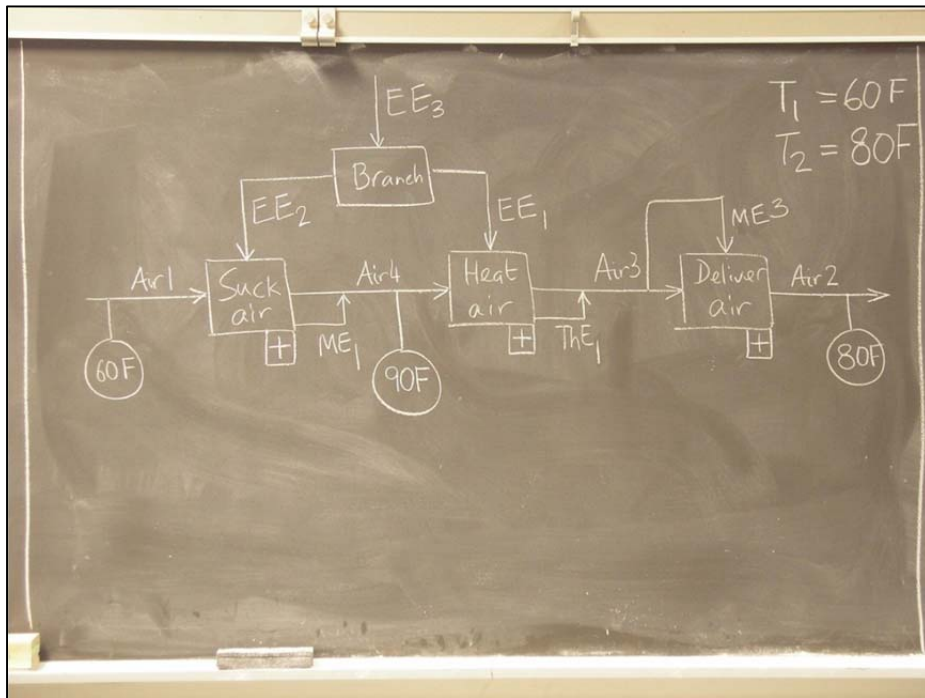
**Chalkboard Exercise Step 28**



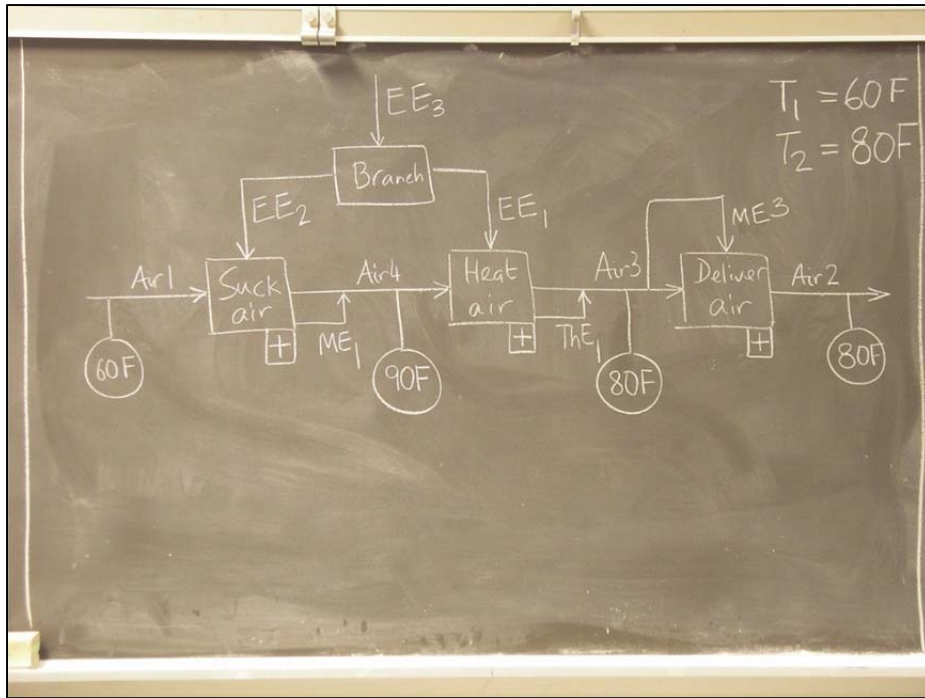
**Chalkboard Exercise Step 29**



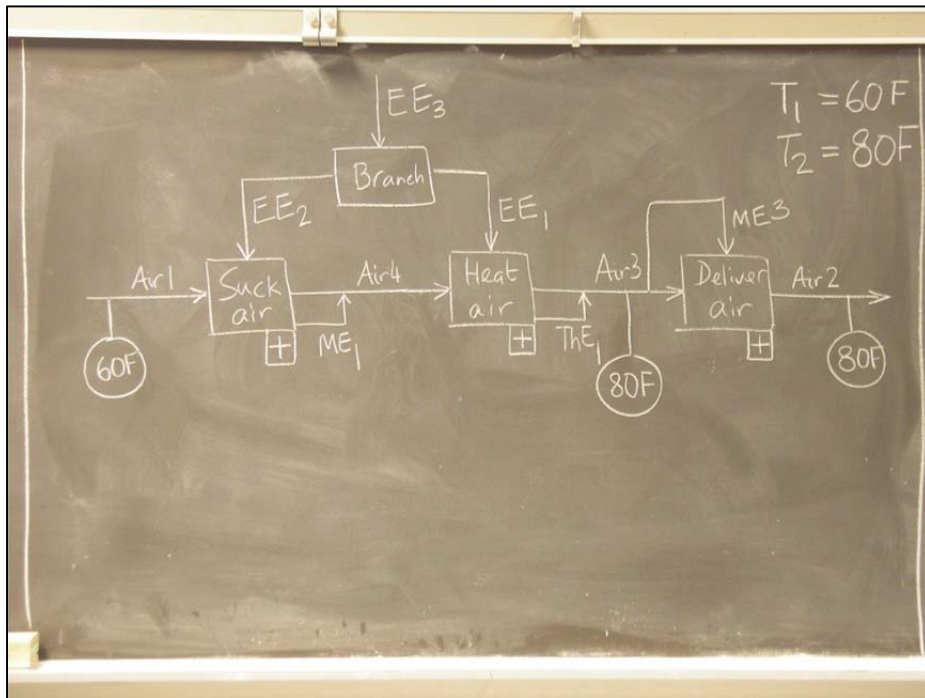
**Chalkboard Exercise Step 30**



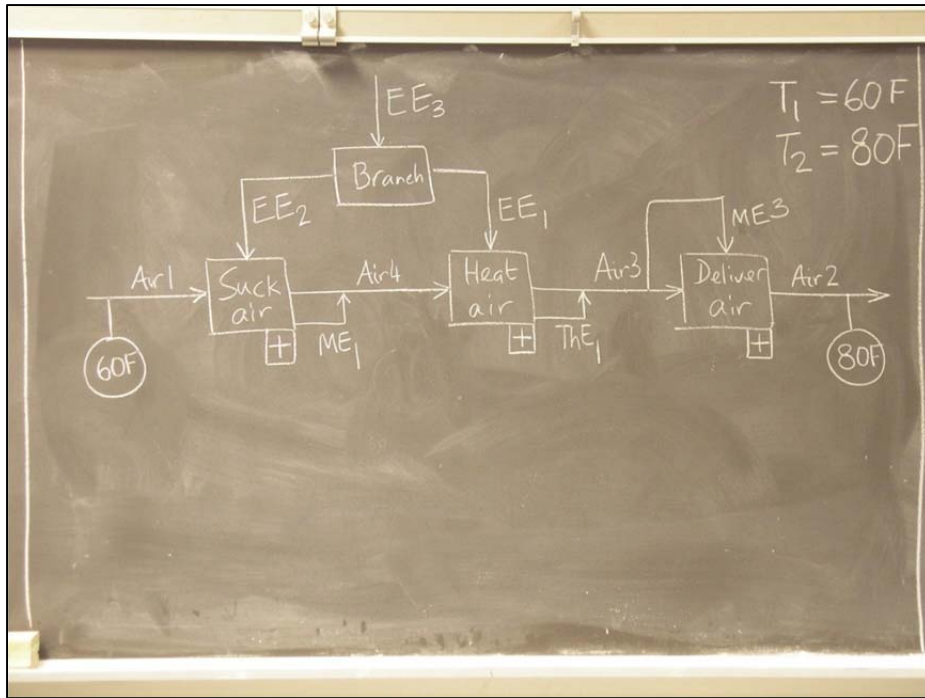
**Chalkboard Exercise Step 31**



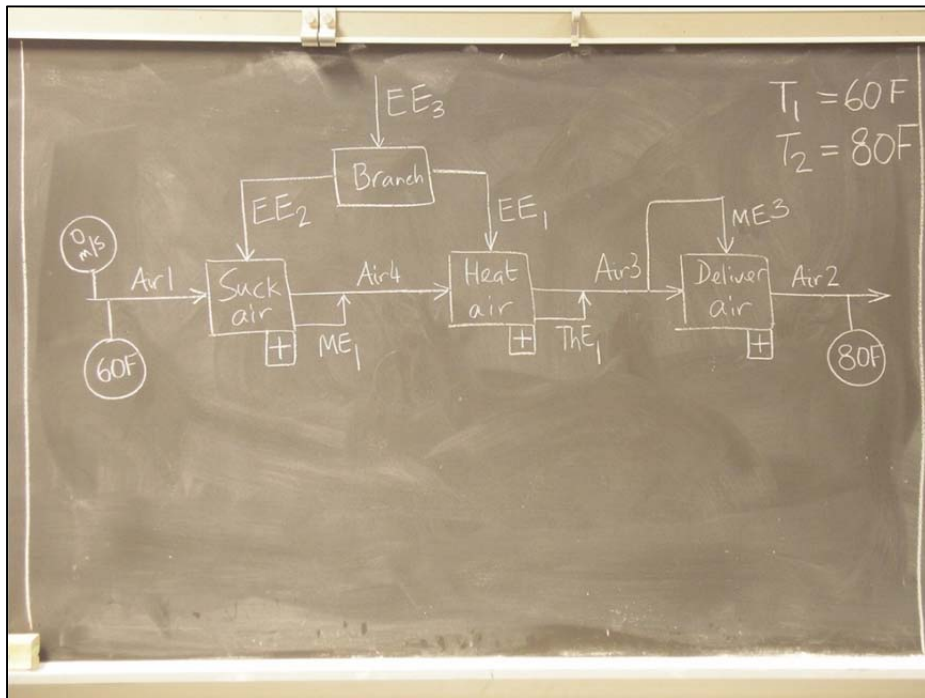
**Chalkboard Exercise Step 32**



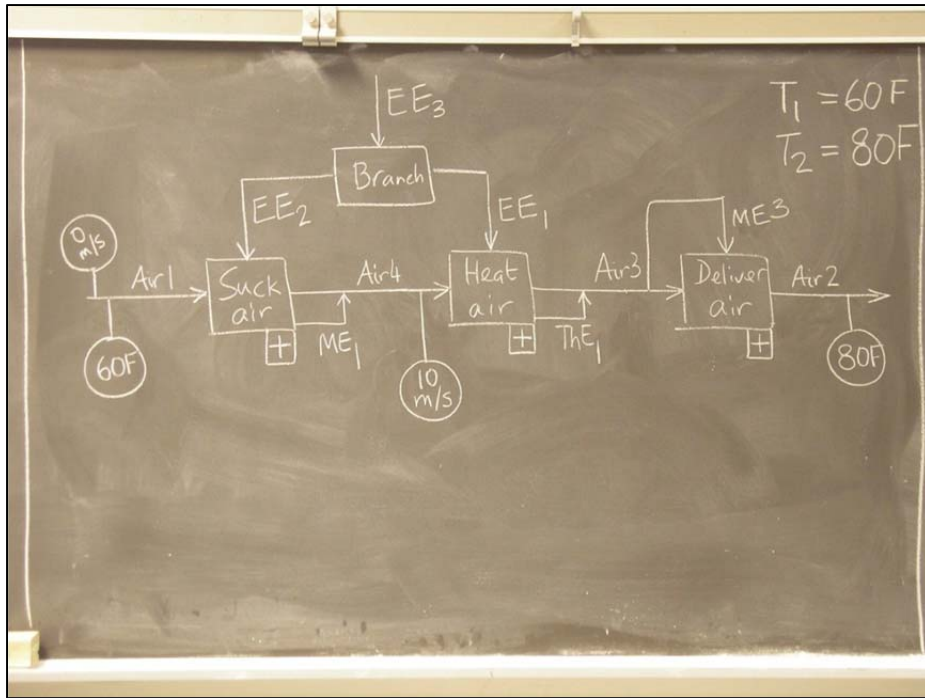
**Chalkboard Exercise Step 33**



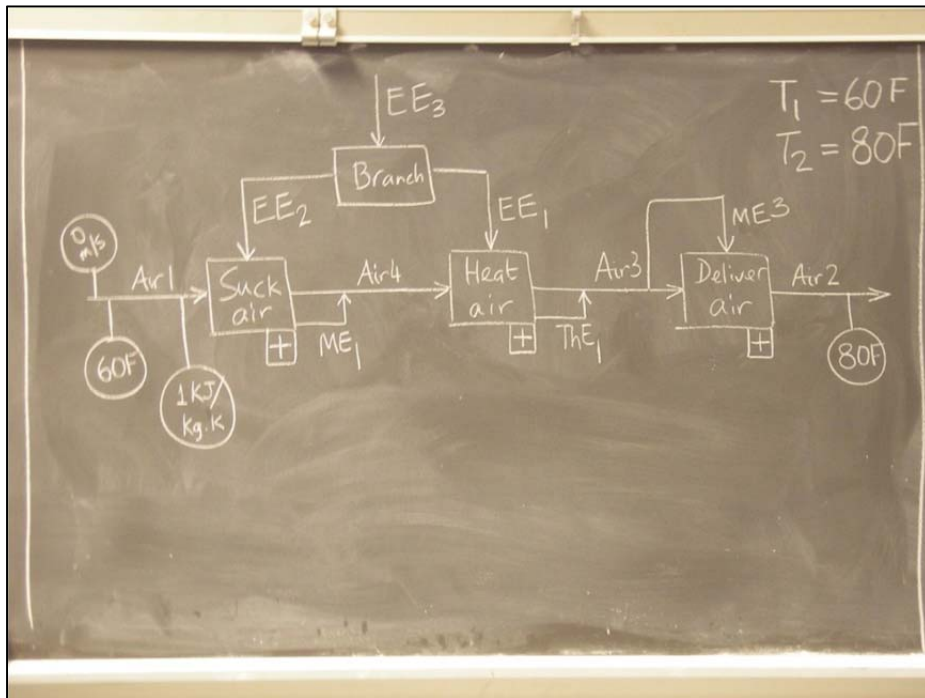
**Chalkboard Exercise Step 34**



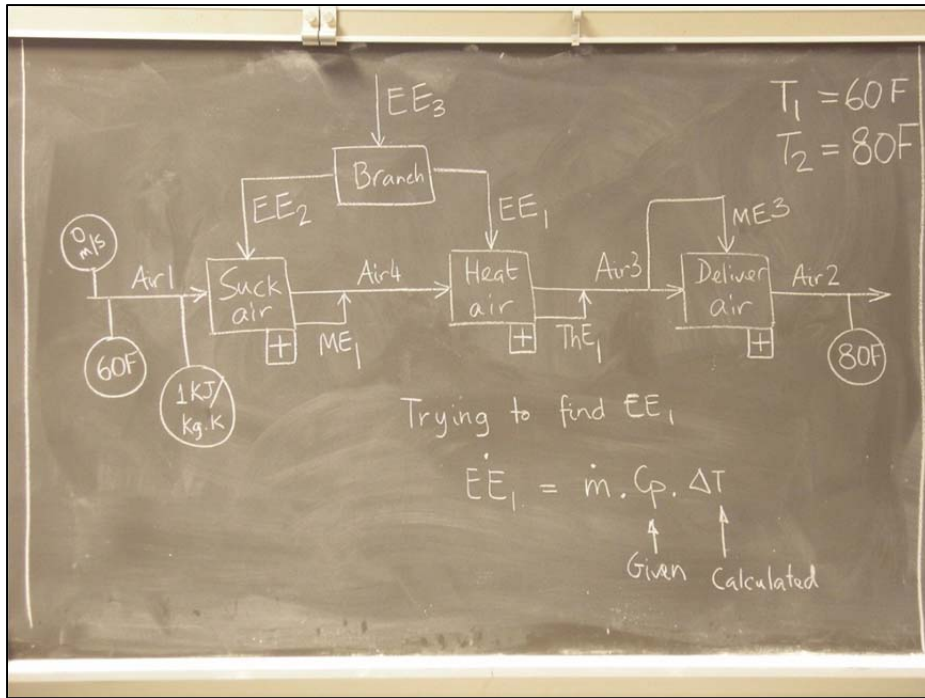
**Chalkboard Exercise Step 35**



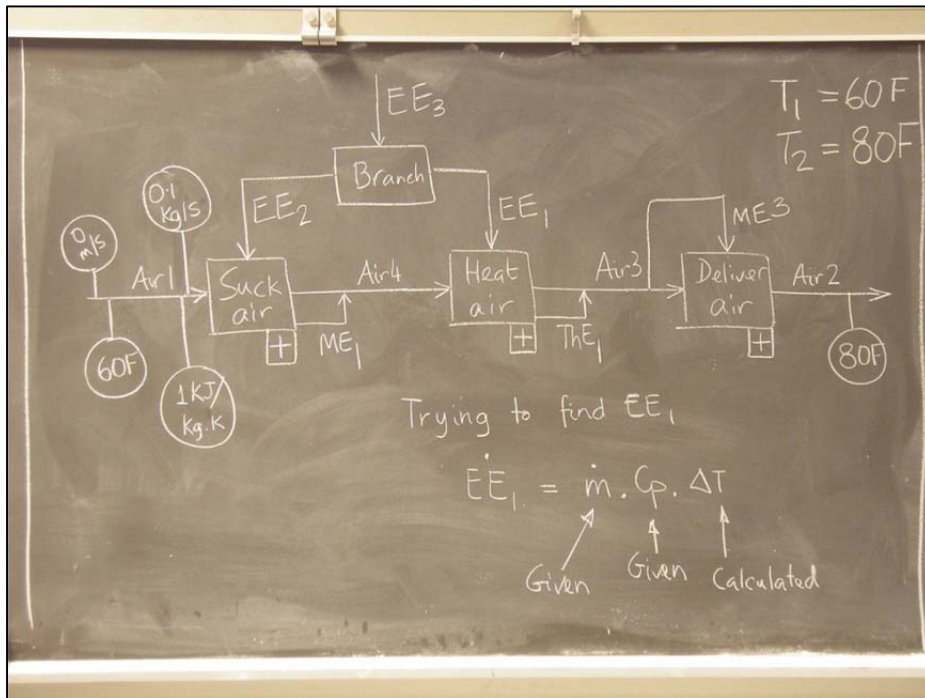
**Chalkboard Exercise Step 36**



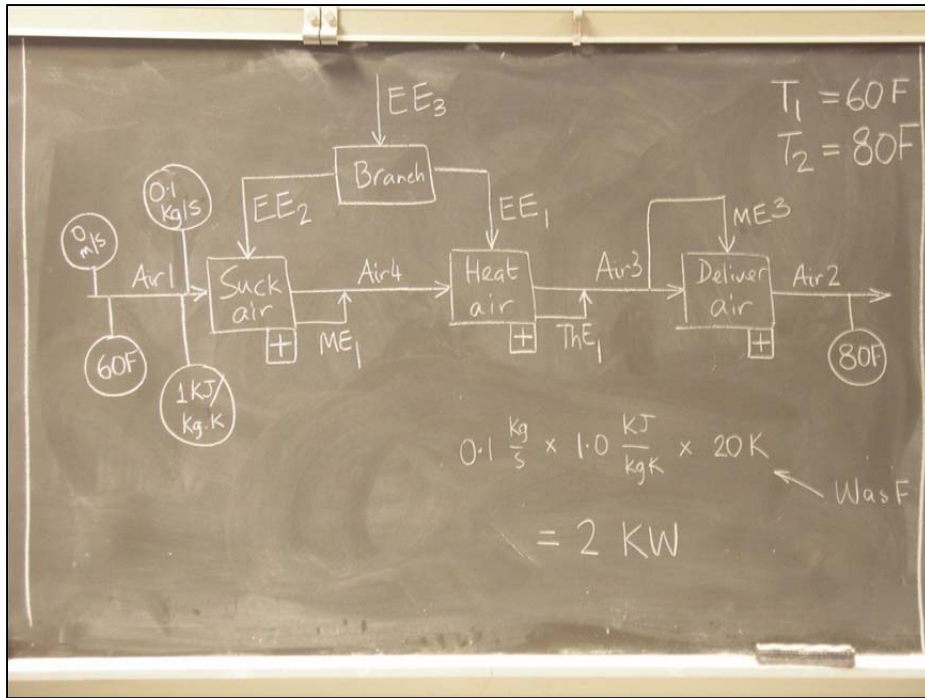
**Chalkboard Exercise Step 37**



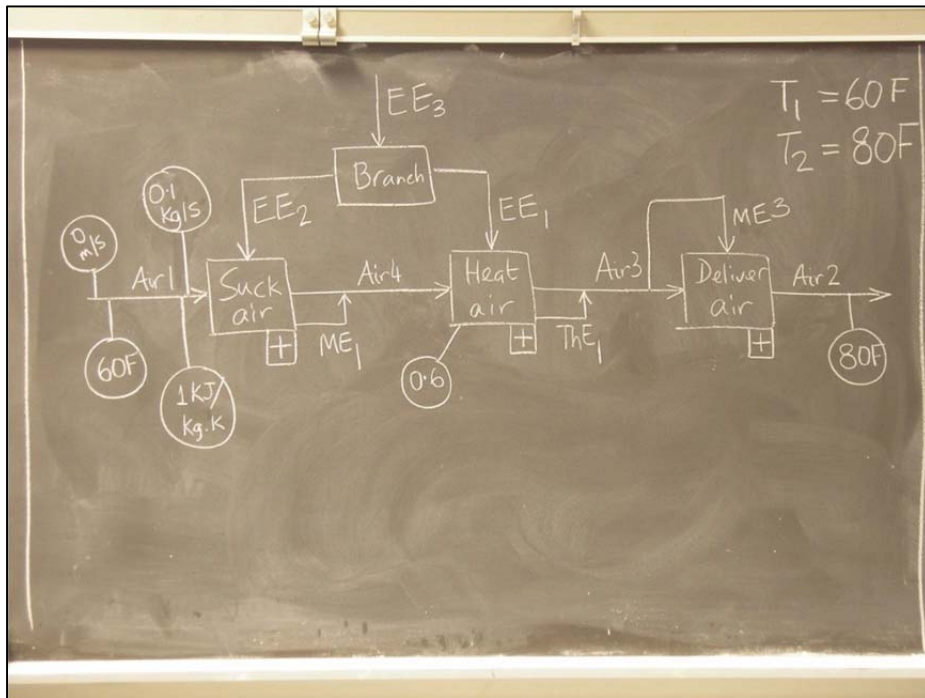
**Chalkboard Exercise Step 38**



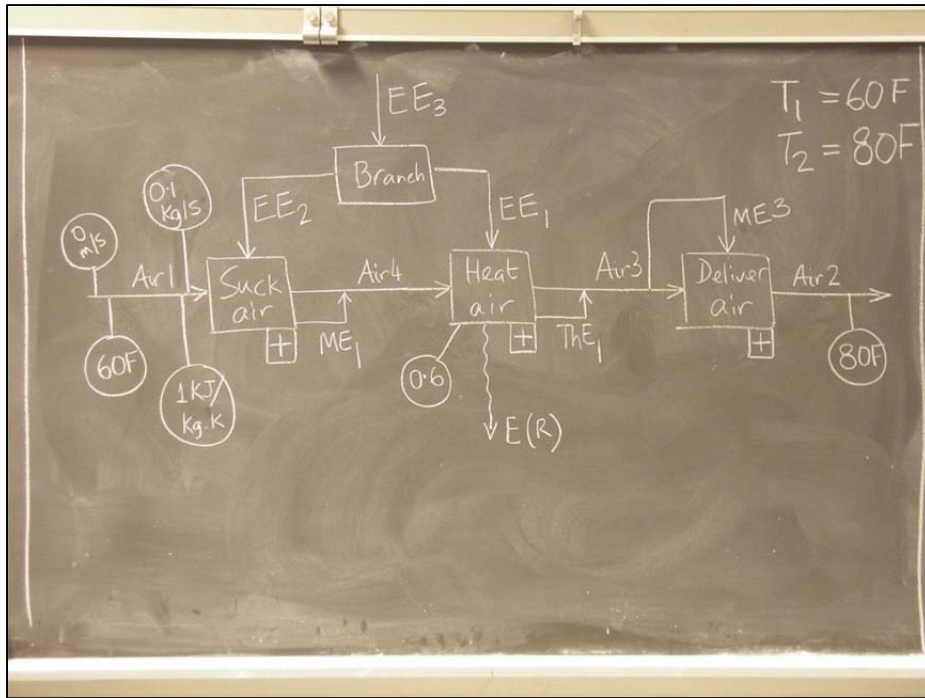
**Chalkboard Exercise Step 39**



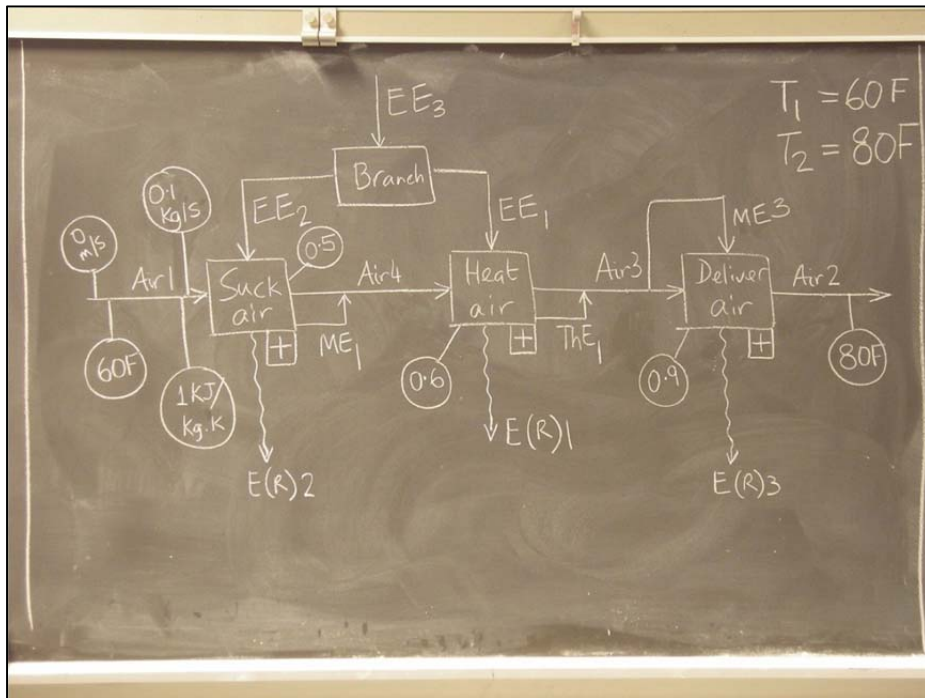
**Chalkboard Exercise Step 40**



**Chalkboard Exercise Step 41**

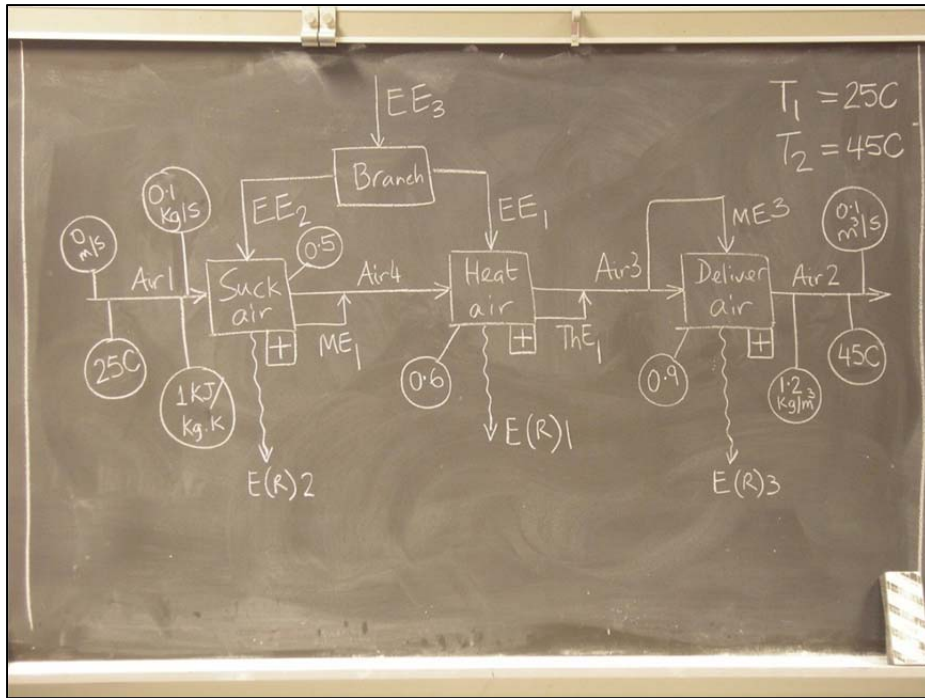


**Chalkboard Exercise Step 42**

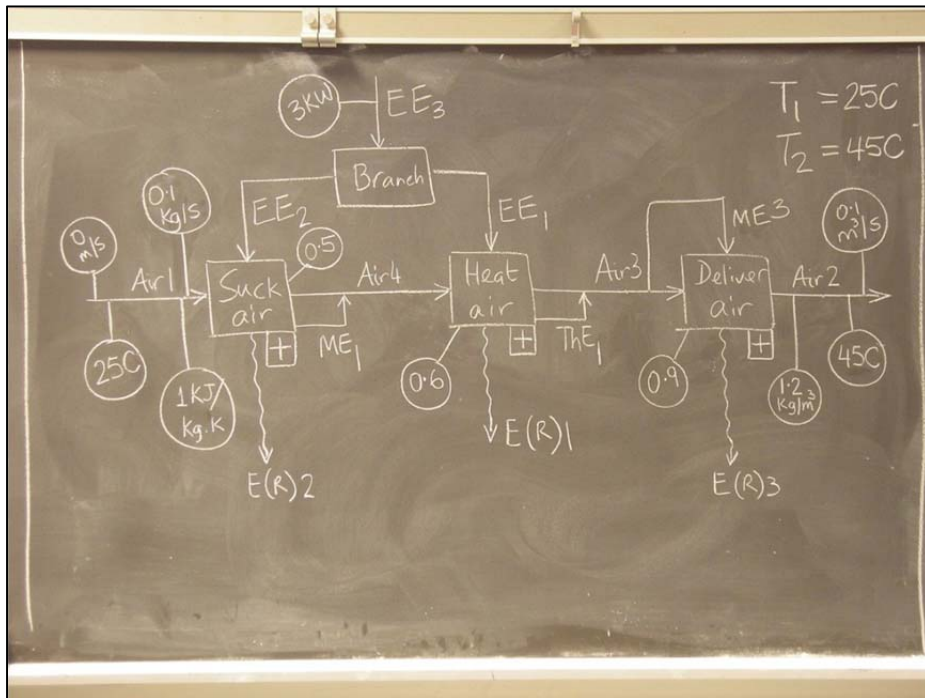


**Chalkboard Exercise Step 43**

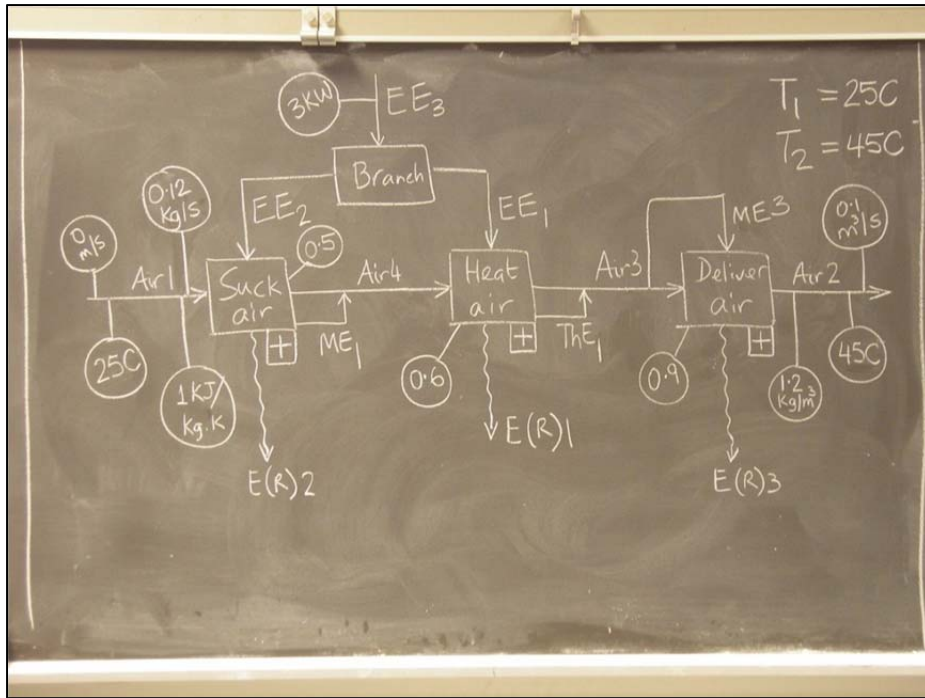




**Chalkboard Exercise Step 44**



**Chalkboard Exercise Step 45**



**Chalkboard Exercise Step 46**

## Appendix B. XML Code for the Function Ontology Presented in Section 6.1.3

The code provided below can be used to reconstruct the ontological view of the representation, including the class definitions, relations (object properties), and attributes (data properties), plus one function structure constructed to illustrate consistency of the representation. The modeled function structure is the air-heating device, first presented in Model State 4.14. The ontology is discussed in Section 6.1.3.

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
  <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
  <!ENTITY swrl "http://www.w3.org/2003/11/swrl#" >
  <!ENTITY swrlb "http://www.w3.org/2003/11/swrlb#" >
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
  <!ENTITY protege "http://protege.stanford.edu/plugins/owl/protege#"
>
  <!ENTITY xsp "http://www.owl-ontologies.com/2005/08/07/xsp.owl#" >
]>

<rdf:RDF xmlns="http://www.owl-ontologies.com/Ontology1310314983.owl#"
  xml:base="http://www.owl-ontologies.com/Ontology1310314983.owl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:xsp="http://www.owl-ontologies.com/2005/08/07/xsp.owl#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="AcE">
    <rdfs:subClassOf rdf:resource="#Energy"/>
    <owl:disjointWith rdf:resource="#ChE"/>
    <owl:disjointWith rdf:resource="#EE"/>
    <owl:disjointWith rdf:resource="#EME"/>
    <owl:disjointWith rdf:resource="#MagE"/>
    <owl:disjointWith rdf:resource="#ME"/>
    <owl:disjointWith rdf:resource="#The"/>
  </owl:Class>
  <Material rdf:ID="Air_1">
    <hasHeadNode rdf:resource="#En_Air_1"/>

```

```

    <hasTailNode rdf:resource="#Env_1"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Material>
<Material rdf:ID="Air_2">
    <hasBaggage_S rdf:resource="#T"/>
    <hasHeadNode rdf:resource="#Env_2"/>
    <hasTailNode rdf:resource="#Transfer_Air"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Material>
<Material rdf:ID="Air_3">
    <hasBaggage_E rdf:resource="#KE_1"/>
    <hasHeadNode rdf:resource="#En_Air_3"/>
    <hasTailNode rdf:resource="#En_Air_1"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Material>
<Material rdf:ID="Air_4">
    <hasBaggage_E rdf:resource="#MW_2"/>
    <hasBaggage_E rdf:resource="#ThE_1"/>
    <hasBaggage_E rdf:resource="#ThE_2"/>
    <hasHeadNode rdf:resource="#Transfer_Air"/>
    <hasTailNode rdf:resource="#En_Air_3"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Material>
<owl:Class rdf:ID="Balance-">
    <rdfs:subClassOf rdf:resource="#Verb"/>
    <owl:disjointWith rdf:resource="#Convert_E"/>
    <owl:disjointWith rdf:resource="#DeEnergize_M"/>
    <owl:disjointWith rdf:resource="#Distribute_E"/>
    <owl:disjointWith rdf:resource="#Energize_M"/>
    <owl:disjointWith rdf:resource="#Balance+"/>
    <owl:disjointWith rdf:resource="#Store_E"/>
    <owl:disjointWith rdf:resource="#Supply_E"/>
    <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<owl:Class rdf:ID="ChE">
    <rdfs:subClassOf rdf:resource="#Energy"/>
    <owl:disjointWith rdf:resource="#AcE"/>
    <owl:disjointWith rdf:resource="#EE"/>
    <owl:disjointWith rdf:resource="#EME"/>
    <owl:disjointWith rdf:resource="#MagE"/>
    <owl:disjointWith rdf:resource="#ME"/>

```

```

    <owl:disjointWith rdf:resource="#The"/>
</owl:Class>
<owl:Class rdf:ID="Conduct_E">
  <rdfs:subClassOf rdf:resource="#Transfer_E"/>
  <owl:disjointWith rdf:resource="#Convect_E"/>
  <owl:disjointWith rdf:resource="#Radiate_E"/>
</owl:Class>
<Verb rdf:ID="Conduct_EE">
  <hasInput rdf:resource="#EE_2"/>
  <hasInput rdf:resource="#T"/>
  <hasOutput rdf:resource="#EE_3"/>
  <hasOutput rdf:resource="#EE_4"/>
</Verb>
<Verb rdf:ID="Conduct_Heat">
  <hasInput rdf:resource="#The_2"/>
  <hasOutput rdf:resource="#Loss_9"/>
</Verb>
<owl:Class rdf:ID="Convect_E">
  <rdfs:subClassOf rdf:resource="#Transfer_E"/>
  <owl:disjointWith rdf:resource="#Conduct_E"/>
  <owl:disjointWith rdf:resource="#Radiate_E"/>
</owl:Class>
<Verb rdf:ID="Convert_1">
  <hasInput rdf:resource="#EE_1"/>
  <hasOutput rdf:resource="#Loss_5"/>
  <hasOutput rdf:resource="#MW_1"/>
</Verb>
<Verb rdf:ID="Convert_2">
  <hasInput rdf:resource="#EE_3"/>
  <hasInput rdf:resource="#EE_4"/>
  <hasOutput rdf:resource="#Loss_7"/>
  <hasOutput rdf:resource="#The_3"/>
</Verb>
<owl:Class rdf:ID="Convert_E">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#DeEnergize_M"/>
  <owl:disjointWith rdf:resource="#Distribute_E"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Balance+"/>
  <owl:disjointWith rdf:resource="#Store_E"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>
  <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<owl:Class rdf:ID="DeEnergize_M">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#Convert_E"/>
  <owl:disjointWith rdf:resource="#Distribute_E"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Balance+"/>
  <owl:disjointWith rdf:resource="#Store_E"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>

```

```

    <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<Verb rdf:ID="Distribute_1">
  <hasOutput rdf:resource="#EE_1"/>
  <hasOutput rdf:resource="#EE_2"/>
</Verb>
<owl:Class rdf:ID="Distribute_E">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#Convert_E"/>
  <owl:disjointWith rdf:resource="#DeEnergize_M"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Balance+"/>
  <owl:disjointWith rdf:resource="#Store_E"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>
  <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<owl:FunctionalProperty rdf:ID="E_hasCarrier">
  <rdf:type rdf:resource="#owl:TransitiveProperty"/>
  <rdf:type rdf:resource="#owl:ObjectProperty"/>
  <rdfs:domain rdf:resource="#Energy"/>
  <owl:inverseOf rdf:resource="#hasBaggage_E"/>
  <rdfs:range rdf:resource="#Material"/>
  <rdfs:subPropertyOf rdf:resource="#hasCarrierFlow"/>
</owl:FunctionalProperty>
<owl:Class rdf:ID="EE">
  <rdfs:subClassOf rdf:resource="#Energy"/>
  <owl:disjointWith rdf:resource="#AcE"/>
  <owl:disjointWith rdf:resource="#ChE"/>
  <owl:disjointWith rdf:resource="#EME"/>
  <owl:disjointWith rdf:resource="#MagE"/>
  <owl:disjointWith rdf:resource="#ME"/>
  <owl:disjointWith rdf:resource="#ThE"/>
</owl:Class>
<Energy rdf:ID="EE_1">
  <hasHeadNode rdf:resource="#Convert_1"/>
  <hasTailNode rdf:resource="#Distribute_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="EE_2">
  <hasHeadNode rdf:resource="#Conduct_EE"/>
  <hasTailNode rdf:resource="#Distribute_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="EE_3">
  <hasHeadNode rdf:resource="#Convert_2"/>
  <hasTailNode rdf:resource="#Conduct_EE"/>

```

```

    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
  </Energy>
  <Energy rdf:ID="EE_4">
    <hasHeadNode rdf:resource="#Convert_2"/>
    <hasTailNode rdf:resource="#Conduct_EE"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
  </Energy>
  <owl:Class rdf:ID="EE_Static">
    <rdfs:subClassOf rdf:resource="#EE"/>
    <owl:disjointWith rdf:resource="#EW"/>
  </owl:Class>
  <owl:Class rdf:ID="EME">
    <rdfs:subClassOf rdf:resource="#Energy"/>
    <owl:disjointWith rdf:resource="#AcE"/>
    <owl:disjointWith rdf:resource="#ChE"/>
    <owl:disjointWith rdf:resource="#EE"/>
    <owl:disjointWith rdf:resource="#MagE"/>
    <owl:disjointWith rdf:resource="#ME"/>
    <owl:disjointWith rdf:resource="#ThE"/>
  </owl:Class>
  <Verb rdf:ID="En_Air_1">
    <hasInput rdf:resource="#Air_1"/>
    <hasInput rdf:resource="#MW_1"/>
    <hasOutput rdf:resource="#Air_3"/>
    <hasOutput rdf:resource="#KE_1"/>
    <hasOutput rdf:resource="#Loss_4"/>
  </Verb>
  <Verb rdf:ID="En_Air_3">
    <hasInput rdf:resource="#Air_3"/>
    <hasInput rdf:resource="#KE_1"/>
    <hasInput rdf:resource="#ThE_3"/>
    <hasOutput rdf:resource="#Air_4"/>
    <hasOutput rdf:resource="#Loss_6"/>
    <hasOutput rdf:resource="#MW_2"/>
    <hasOutput rdf:resource="#ThE_1"/>
    <hasOutput rdf:resource="#ThE_2"/>
  </Verb>
  <owl:Class rdf:ID="Energize_M">
    <rdfs:subClassOf rdf:resource="#Verb"/>
    <owl:disjointWith rdf:resource="#Balance-"/>
    <owl:disjointWith rdf:resource="#Convert_E"/>
    <owl:disjointWith rdf:resource="#DeEnergize_M"/>
    <owl:disjointWith rdf:resource="#Distribute_E"/>
    <owl:disjointWith rdf:resource="#Balance+"/>
    <owl:disjointWith rdf:resource="#Store_E"/>
    <owl:disjointWith rdf:resource="#Supply_E"/>
    <owl:disjointWith rdf:resource="#Transfer_E"/>

```

```

</owl:Class>
<owl:Class rdf:ID="Energy">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasTailNode"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Source"/>
            <owl:Class rdf:about="#Verb"/>
          </owl:unionOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Noun"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#E_hasCarrier"/>
      <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Material"/>
  <owl:disjointWith rdf:resource="#Signal"/>
</owl:Class>
<owl:Class rdf:ID="Env">
  <rdfs:subClassOf rdf:resource="#Node"/>
  <owl:disjointWith rdf:resource="#Verb"/>
</owl:Class>
<Source rdf:ID="Env_1">
  <hasOutput rdf:resource="#Air_1"/>
</Source>
<Sink rdf:ID="Env_2">
  <hasInput rdf:resource="#Air_2"/>
</Sink>
<Source rdf:ID="Env_3">
<Sink rdf:ID="Env_4">
  <hasInput rdf:resource="#Loss_4"/>
  <hasInput rdf:resource="#Loss_5"/>
</Sink>
<Sink rdf:ID="Env_5">
  <hasInput rdf:resource="#Loss_6"/>
  <hasInput rdf:resource="#Loss_7"/>
</Sink>
<Sink rdf:ID="Env_6">
  <hasInput rdf:resource="#Loss_8"/>
  <hasInput rdf:resource="#Loss_9"/>
</Sink>
<owl:Class rdf:ID="EW">
  <rdfs:subClassOf rdf:resource="#EE"/>
  <owl:disjointWith rdf:resource="#EE_Static"/>
</owl:Class>

```



```

<owl:Class rdf:ID="Gas">
  <rdfs:subClassOf rdf:resource="#Gaseous"/>
  <owl:disjointWith rdf:resource="#Vapor"/>
</owl:Class>
<owl:Class rdf:ID="Gaseous">
  <rdfs:subClassOf rdf:resource="#Material"/>
  <owl:disjointWith rdf:resource="#Liquid"/>
  <owl:disjointWith rdf:resource="#Solid"/>
</owl:Class>
<owl:FunctionalProperty rdf:ID="GeometricCenter_X">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="GeometricCenter_Y">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="GivenName">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:FunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="hasBaggage_E">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain rdf:resource="#Material"/>
  <owl:inverseOf rdf:resource="#E_hasCarrier"/>
  <rdfs:range rdf:resource="#Energy"/>
  <rdfs:subPropertyOf rdf:resource="#hasBaggageFlow"/>
</owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="hasBaggage_S">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Energy"/>
        <owl:Class rdf:about="#Material"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <owl:inverseOf rdf:resource="#S_hasCarrier"/>
  <rdfs:range rdf:resource="#Signal"/>
  <rdfs:subPropertyOf rdf:resource="#hasBaggageFlow"/>
</owl:InverseFunctionalProperty>
<owl:ObjectProperty rdf:ID="hasBaggageFlow">
  <owl:inverseOf rdf:resource="#hasCarrierFlow"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasCarrierFlow">
  <owl:inverseOf rdf:resource="#hasBaggageFlow"/>
</owl:ObjectProperty>
<owl:TransitiveProperty rdf:ID="hasChild">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <owl:inverseOf rdf:resource="#hasParent"/>
</owl:TransitiveProperty>
<owl:TransitiveProperty rdf:ID="hasChild_E">

```

```

    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain rdf:resource="#Energy"/>
    <owl:inverseOf rdf:resource="#hasParent_E"/>
    <rdfs:range rdf:resource="#Energy"/>
    <rdfs:subPropertyOf rdf:resource="#hasChild"/>
</owl:TransitiveProperty>
<owl:TransitiveProperty rdf:ID="hasChild_M">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain rdf:resource="#Material"/>
    <owl:inverseOf rdf:resource="#hasParent_M"/>
    <rdfs:range rdf:resource="#Material"/>
    <rdfs:subPropertyOf rdf:resource="#hasChild"/>
</owl:TransitiveProperty>
<owl:FunctionalProperty rdf:ID="hasHeadNode">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain rdf:resource="#Noun"/>
    <owl:inverseOf rdf:resource="#hasInput"/>
    <rdfs:range>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Sink"/>
                <owl:Class rdf:about="#Verb"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:range>
    <rdfs:subPropertyOf rdf:resource="#hasTerminal"/>
</owl:FunctionalProperty>
<owl:ObjectProperty rdf:ID="hasInOutFlows">
    <owl:inverseOf rdf:resource="#hasTerminal"/>
</owl:ObjectProperty>
<owl:InverseFunctionalProperty rdf:ID="hasInput">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Sink"/>
                <owl:Class rdf:about="#Verb"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:domain>
    <owl:inverseOf rdf:resource="#hasHeadNode"/>
    <rdfs:range rdf:resource="#Noun"/>
    <rdfs:subPropertyOf rdf:resource="#hasInOutFlows"/>
</owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="hasOutput">
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Source"/>
                <owl:Class rdf:about="#Verb"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:domain>

```

```

</rdfs:domain>
<owl:inverseOf rdf:resource="#hasTailNode"/>
<rdfs:range>
  <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Energy"/>
      <owl:Class rdf:about="#Material"/>
    </owl:unionOf>
  </owl:Class>
</rdfs:range>
<rdfs:subPropertyOf rdf:resource="#hasInOutFlows"/>
</owl:InverseFunctionalProperty>
<owl:TransitiveProperty rdf:ID="hasParent">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <owl:inverseOf rdf:resource="#hasChild"/>
</owl:TransitiveProperty>
<owl:TransitiveProperty rdf:ID="hasParent_E">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain rdf:resource="#Energy"/>
  <owl:inverseOf rdf:resource="#hasChild_E"/>
  <rdfs:range rdf:resource="#Energy"/>
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
</owl:TransitiveProperty>
<owl:TransitiveProperty rdf:ID="hasParent_M">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain rdf:resource="#Material"/>
  <owl:inverseOf rdf:resource="#hasChild_M"/>
  <rdfs:range rdf:resource="#Material"/>
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
</owl:TransitiveProperty>
<owl:FunctionalProperty rdf:ID="hasTailNode">
  <rdf:type rdf:resource="&owl;ObjectProperty"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Energy"/>
        <owl:Class rdf:about="#Material"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <owl:inverseOf rdf:resource="#hasOutput"/>
<rdfs:range>
  <owl:Class>
    <owl:unionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Source"/>
      <owl:Class rdf:about="#Verb"/>
    </owl:unionOf>
  </owl:Class>
</rdfs:range>
<rdfs:subPropertyOf rdf:resource="#hasTerminal"/>
</owl:FunctionalProperty>
<owl:ObjectProperty rdf:ID="hasTerminal">
  <owl:inverseOf rdf:resource="#hasInOutFlows"/>

```

```

</owl:ObjectProperty>
<owl:FunctionalProperty rdf:ID="HeadPoint_X">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:domain rdf:resource="#Noun"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="HeadPoint_Y">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:domain rdf:resource="#Noun"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:Class rdf:ID="Balance+">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#Convert_E"/>
  <owl:disjointWith rdf:resource="#DeEnergize_M"/>
  <owl:disjointWith rdf:resource="#Distribute_E"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Store_E"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>
  <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<owl:Class rdf:ID="KE">
  <rdfs:subClassOf rdf:resource="#ME"/>
  <owl:disjointWith rdf:resource="#MW"/>
  <owl:disjointWith rdf:resource="#PE"/>
</owl:Class>
<Energy rdf:ID="KE_1">
  <E_hasCarrier rdf:resource="#Air_3"/>
  <hasHeadNode rdf:resource="#En_Air_3"/>
  <hasTailNode rdf:resource="#En_Air_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<owl:Class rdf:ID="KE_Linear">
  <rdfs:subClassOf rdf:resource="#KE"/>
  <owl:disjointWith rdf:resource="#KE_Rotational"/>
</owl:Class>
<owl:Class rdf:ID="KE_Rotational">
  <rdfs:subClassOf rdf:resource="#KE"/>
  <owl:disjointWith rdf:resource="#KE_Linear"/>
</owl:Class>
<owl:Class rdf:ID="Liquid">
  <rdfs:subClassOf rdf:resource="#Material"/>
  <owl:disjointWith rdf:resource="#Gaseous"/>
  <owl:disjointWith rdf:resource="#Solid"/>
</owl:Class>
<Energy rdf:ID="Loss_4">
  <hasHeadNode rdf:resource="#Env_4"/>
  <hasTailNode rdf:resource="#En_Air_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>

```

```

    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="Loss_5">
  <hasHeadNode rdf:resource="#Env_4"/>
  <hasTailNode rdf:resource="#Convert_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="Loss_6">
  <hasHeadNode rdf:resource="#Env_5"/>
  <hasTailNode rdf:resource="#En_Air_3"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="Loss_7">
  <hasHeadNode rdf:resource="#Env_5"/>
  <hasTailNode rdf:resource="#Convert_2"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="Loss_8">
  <hasHeadNode rdf:resource="#Env_6"/>
  <hasTailNode rdf:resource="#Transfer_Air"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="Loss_9">
  <hasHeadNode rdf:resource="#Env_6"/>
  <hasTailNode rdf:resource="#Conduct_Heat"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<owl:Class rdf:ID="MagE">
  <rdfs:subClassOf rdf:resource="#Energy"/>
  <owl:disjointWith rdf:resource="#AcE"/>
  <owl:disjointWith rdf:resource="#ChE"/>
  <owl:disjointWith rdf:resource="#EE"/>
  <owl:disjointWith rdf:resource="#EME"/>
  <owl:disjointWith rdf:resource="#ME"/>
  <owl:disjointWith rdf:resource="#ThE"/>
</owl:Class>

```

```

<owl:Class rdf:ID="Material">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasTailNode"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Source"/>
            <owl:Class rdf:about="#Verb"/>
          </owl:unionOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Noun"/>
  <owl:disjointWith rdf:resource="#Energy"/>
  <owl:disjointWith rdf:resource="#Signal"/>
</owl:Class>
<owl:Class rdf:ID="ME">
  <rdfs:subClassOf rdf:resource="#Energy"/>
  <owl:disjointWith rdf:resource="#AcE"/>
  <owl:disjointWith rdf:resource="#ChE"/>
  <owl:disjointWith rdf:resource="#EE"/>
  <owl:disjointWith rdf:resource="#EME"/>
  <owl:disjointWith rdf:resource="#MagE"/>
  <owl:disjointWith rdf:resource="#ThE"/>
</owl:Class>
<owl:Class rdf:ID="MW">
  <rdfs:subClassOf rdf:resource="#ME"/>
  <owl:disjointWith rdf:resource="#KE"/>
  <owl:disjointWith rdf:resource="#PE"/>
</owl:Class>
<Energy rdf:ID="MW_1">
  <hasHeadNode rdf:resource="#En_Air_1"/>
  <hasTailNode rdf:resource="#Convert_1"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="MW_2">
  <E_hasCarrier rdf:resource="#Air_4"/>
  <hasHeadNode rdf:resource="#Transfer_Air"/>
  <hasTailNode rdf:resource="#En_Air_3"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<owl:Class rdf:ID="Node">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasInOutFlows"/>

```

```

        <owl:someValuesFrom>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <owl:Class rdf:about="#Energy"/>
                    <owl:Class rdf:about="#Material"/>
                </owl:unionOf>
            </owl:Class>
        </owl:someValuesFrom>
    </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf rdf:resource="&owl;Thing"/>
<owl:disjointWith rdf:resource="#Noun"/>
</owl:Class>
<owl:Class rdf:ID="Noun">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasHeadNode"/>
            <owl:someValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <owl:Class rdf:about="#Sink"/>
                        <owl:Class rdf:about="#Verb"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:someValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>
<rdfs:subClassOf rdf:resource="&owl;Thing"/>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#HeadPoint_X"/>
        <owl:cardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#HeadPoint_Y"/>
        <owl:cardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#TailPoint_X"/>
        <owl:cardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
    </owl:Restriction>
</rdfs:subClassOf>
<rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#TailPoint_Y"/>

```

```

        <owl:cardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Node"/>
</owl:Class>
<owl:Class rdf:ID="PE">
    <rdfs:subClassOf rdf:resource="#ME"/>
    <owl:disjointWith rdf:resource="#KE"/>
    <owl:disjointWith rdf:resource="#MW"/>
</owl:Class>
<owl:Class rdf:ID="PE_Elastic">
    <rdfs:subClassOf rdf:resource="#PE"/>
    <owl:disjointWith rdf:resource="#PE_Gravitational"/>
</owl:Class>
<owl:Class rdf:ID="PE_Gravitational">
    <rdfs:subClassOf rdf:resource="#PE"/>
    <owl:disjointWith rdf:resource="#PE_Elastic"/>
</owl:Class>
<owl:Class rdf:ID="Radiate_E">
    <rdfs:subClassOf rdf:resource="#Transfer_E"/>
    <owl:disjointWith rdf:resource="#Conduct_E"/>
    <owl:disjointWith rdf:resource="#Convect_E"/>
</owl:Class>
<owl:FunctionalProperty rdf:ID="S_hasCarrier">
    <rdf:type rdf:resource="&owl;TransitiveProperty"/>
    <rdf:type rdf:resource="&owl;ObjectProperty"/>
    <rdfs:domain rdf:resource="#Signal"/>
    <owl:inverseOf rdf:resource="#hasBaggage_S"/>
    <rdfs:range>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Energy"/>
                <owl:Class rdf:about="#Material"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:range>
    <rdfs:subPropertyOf rdf:resource="#hasCarrierFlow"/>
</owl:FunctionalProperty>
<owl:Class rdf:ID="Signal">
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#S_hasCarrier"/>
            <owl:someValuesFrom>
                <owl:Class>
                    <owl:unionOf rdf:parseType="Collection">
                        <owl:Class rdf:about="#Energy"/>
                        <owl:Class rdf:about="#Material"/>
                    </owl:unionOf>
                </owl:Class>
            </owl:someValuesFrom>
        </owl:Restriction>
    </rdfs:subClassOf>

```



```

    <rdfs:subClassOf rdf:resource="#Noun"/>
    <owl:disjointWith rdf:resource="#Energy"/>
    <owl:disjointWith rdf:resource="#Material"/>
</owl:Class>
<owl:Class rdf:ID="Sink">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasInput"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Energy"/>
            <owl:Class rdf:about="#Material"/>
          </owl:unionOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Env"/>
</owl:Class>
<owl:Class rdf:ID="Solid">
  <rdfs:subClassOf rdf:resource="#Material"/>
  <owl:disjointWith rdf:resource="#Gaseous"/>
  <owl:disjointWith rdf:resource="#Liquid"/>
</owl:Class>
<owl:Class rdf:ID="Source">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasOutput"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Energy"/>
            <owl:Class rdf:about="#Material"/>
          </owl:unionOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Env"/>
</owl:Class>
<owl:Class rdf:ID="Store_E">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#Convert_E"/>
  <owl:disjointWith rdf:resource="#DeEnergize_M"/>
  <owl:disjointWith rdf:resource="#Distribute_E"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Balance+"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>
  <owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<owl:Class rdf:ID="Supply_E">

```

```

<rdfs:subClassOf rdf:resource="#Verb"/>
<owl:disjointWith rdf:resource="#Balance-"/>
<owl:disjointWith rdf:resource="#Convert_E"/>
<owl:disjointWith rdf:resource="#DeEnergize_M"/>
<owl:disjointWith rdf:resource="#Distribute_E"/>
<owl:disjointWith rdf:resource="#Energize_M"/>
<owl:disjointWith rdf:resource="#Balance+"/>
<owl:disjointWith rdf:resource="#Store_E"/>
<owl:disjointWith rdf:resource="#Transfer_E"/>
</owl:Class>
<Signal rdf:ID="T">
  <hasHeadNode rdf:resource="#Conduct_EE"/>
  <S_hasCarrier rdf:resource="#Air_2"/>
</Signal>
<owl:FunctionalProperty rdf:ID="TailPoint_X">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:domain rdf:resource="#Noun"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="TailPoint_Y">
  <rdf:type rdf:resource="&owl;DatatypeProperty"/>
  <rdfs:domain rdf:resource="#Noun"/>
  <rdfs:range rdf:resource="&xsd;float"/>
</owl:FunctionalProperty>
<owl:Class rdf:ID="TheE">
  <rdfs:subClassOf rdf:resource="#Energy"/>
  <owl:disjointWith rdf:resource="#AcE"/>
  <owl:disjointWith rdf:resource="#ChE"/>
  <owl:disjointWith rdf:resource="#EE"/>
  <owl:disjointWith rdf:resource="#EME"/>
  <owl:disjointWith rdf:resource="#MagE"/>
  <owl:disjointWith rdf:resource="#ME"/>
</owl:Class>
<Energy rdf:ID="The_1">
  <E_hasCarrier rdf:resource="#Air_4"/>
  <hasHeadNode rdf:resource="#Transfer_Air"/>
  <hasTailNode rdf:resource="#En_Air_3"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="The_2">
  <E_hasCarrier rdf:resource="#Air_4"/>
  <hasHeadNode rdf:resource="#Conduct_Heat"/>
  <hasTailNode rdf:resource="#En_Air_3"/>
  <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
  <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
  <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
  <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<Energy rdf:ID="The_3">
  <hasHeadNode rdf:resource="#En_Air_3"/>

```

```

    <hasTailNode rdf:resource="#Convert_2"/>
    <HeadPoint_X rdf:datatype="&xsd;float">0.0</HeadPoint_X>
    <HeadPoint_Y rdf:datatype="&xsd;float">0.0</HeadPoint_Y>
    <TailPoint_X rdf:datatype="&xsd;float">0.0</TailPoint_X>
    <TailPoint_Y rdf:datatype="&xsd;float">0.0</TailPoint_Y>
</Energy>
<owl:Class rdf:ID="ThE_Latent">
  <rdfs:subClassOf rdf:resource="#ThE"/>
  <owl:disjointWith rdf:resource="#ThE_Sensible"/>
</owl:Class>
<owl:Class rdf:ID="ThE_Sensible">
  <rdfs:subClassOf rdf:resource="#ThE"/>
  <owl:disjointWith rdf:resource="#ThE_Latent"/>
</owl:Class>
<Verb rdf:ID="Transfer_Air">
  <hasInput rdf:resource="#Air_4"/>
  <hasInput rdf:resource="#MW_2"/>
  <hasInput rdf:resource="#ThE_1"/>
  <hasOutput rdf:resource="#Air_2"/>
  <hasOutput rdf:resource="#Loss_8"/>
</Verb>
<owl:Class rdf:ID="Transfer_E">
  <rdfs:subClassOf rdf:resource="#Verb"/>
  <owl:disjointWith rdf:resource="#Balance-"/>
  <owl:disjointWith rdf:resource="#Convert_E"/>
  <owl:disjointWith rdf:resource="#DeEnergize_M"/>
  <owl:disjointWith rdf:resource="#Distribute_E"/>
  <owl:disjointWith rdf:resource="#Energize_M"/>
  <owl:disjointWith rdf:resource="#Balance+"/>
  <owl:disjointWith rdf:resource="#Store_E"/>
  <owl:disjointWith rdf:resource="#Supply_E"/>
</owl:Class>
<owl:Class rdf:ID="Vapor">
  <rdfs:subClassOf rdf:resource="#Gaseous"/>
  <owl:disjointWith rdf:resource="#Gas"/>
</owl:Class>
<owl:Class rdf:ID="Verb">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasInput"/>
      <owl:someValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Energy"/>
            <owl:Class rdf:about="#Material"/>
          </owl:unionOf>
        </owl:Class>
      </owl:someValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasOutput"/>

```

```
        <owl:someValuesFrom>
          <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
              <owl:Class rdf:about="#Energy"/>
              <owl:Class rdf:about="#Material"/>
            </owl:unionOf>
          </owl:Class>
        </owl:someValuesFrom>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#Node"/>
    <owl:disjointWith rdf:resource="#Env"/>
  </owl:Class>
</rdf:RDF>
```

## Appendix C. Header Files for the ConMod Application

```
// ChildFrm.h : interface of the CChildFrame class
//

#pragma once

class CChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CChildFrame)
public:
    CChildFrame();

    // Attributes
protected:
    CSplitterWnd m_wndSplitter;
public:

    // Operations
public:

    // Overrides
    public:
        virtual BOOL OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext*
pContext);
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);

    // Implementation
public:
        virtual ~CChildFrame();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

    // Generated message map functions
protected:
        DECLARE_MESSAGE_MAP()
};
```

```
#pragma once
#include "Template.h"

// CConduct_E_Template dialog
```

```

class CConduct_E_Template : public CDialog, public CTemplate
{
    DECLARE_DYNAMIC(CConduct_E_Template)

public:
    CConduct_E_Template(CWnd* pParent = NULL, CPoint InsertionPoint =
(500,500),
        CString*      pCounterString_F      =      NULL,      CString*
pCounterString_InE = NULL,
        CString*      pCounterString_OutE    =      NULL,      CString*
pCounterString_OutE_Res = NULL);

    virtual ~CConduct_E_Template();

// Dialog Data
    enum { IDD = IDD_Conduct_E_TEMPLATE };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

    DECLARE_MESSAGE_MAP()

public:    // Instances that comprise the Convert_E template
    CFunction* pFunctionBlock;
    CEnergy* pEnergy_InE;
    CEnergy* pEnergy_OutE;
    CEnergy* pEnergy_OutE_Res;
};

```

```

// ConMod.h : main header file for the ConMod application
//
#pragma once

#ifndef __AFXWIN_H__
    #error "include 'stdafx.h' before including this file for PCH"
#endif

#include "resource.h"    // main symbols

// CConModApp:
// See ConMod.cpp for the implementation of this class
//
class CConModApp : public CWinApp
{

```

```

public:
    CConModApp();

// Overrides
public:
    virtual BOOL InitInstance();

// Implementation
    afx_msg void OnAppAbout();
    DECLARE_MESSAGE_MAP()
};

extern CConModApp theApp;

```

```

#include "Function.h"
#include "Env.h"
#include "Edge.h"
#include "Material.h"
#include "Energy.h"
#include "Signal.h"
#include "Convert_E_Template.h"
#include "Conduct_E_Template.h"
#include "Energize_M_Template.h"
#include "Distribute_E_Template.h"
#include "DeEn_M_Template.h"

#pragma once

class CConModDoc : public CDocument
{
protected: // create from serialization only
    CConModDoc();
    DECLARE_DYNCREATE(CConModDoc)

// Attributes
public:

// Operations
public:

// Overrides
public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);

// Implementation
public:
    virtual ~CConModDoc();

```

```

#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    DECLARE_MESSAGE_MAP()

public:
    CList<CElement*, CElement*> CElementList;        // list of all
elements of all types - reconstructed everytime OnDraw is called
    CList<CNode*, CNode*> CNodeList;        // List of Function blocks -
appended upon ADD_FUNCTION, removed upon DELETE
    CList<CEdge*, CEdge*> CEdgeList;        // List of
flow arrows (edges) of all kinds - appended upon ADD_EDGE, removed upon
DELETE
    CList<CElement*, CElement*> PreselectionList;

    CList<CFunction*, CFunction*> CFunctionList;
    CList<CEnv*, CEnv*> CEnvList;
    CList<CMaterial*, CMaterial*> CMaterialList;
    CList<CMaterial*, CMaterial*> CMaterialList_IN_TEMP;
    CList<CMaterial*, CMaterial*> CMaterialList_OUT_TEMP;
    CList<CEnergy*, CEnergy*> CEnergyList;
    CList<CEnergy*, CEnergy*> CEnergyList_IN_TEMP;
    CList<CEnergy*, CEnergy*> CEnergyList_OUT_TEMP;
    CList<CSignal*, CSignal*> CSignalList;
    CList<CSignal*, CSignal*> CSignalList_IN_TEMP;
    CList<CSignal*, CSignal*> CSignalList_OUT_TEMP;

    CList<CTemplate*, CTemplate*> CTemplateList;    // List of all
templates of Layer 2
    // The main purpose of this list is to store the "template"
instances, while the individual
    // elements in the templates, such as functions and flows, are
stored in the CElementList.
    // By storing the templates in this separate list, it will be
easier to delete them
    // during application exit (Destructor of the View class).

    CList<CFunction*, CFunction*> CConvert_E_Function_List;
    CList<CConvert_E_Template*,
CConvert_E_Template*>
CConvert_E_Template_List;

    CList<CFunction*, CFunction*> CConduct_E_Function_List;
    CList<CConduct_E_Template*,
CConduct_E_Template*>
CConduct_E_Template_List;

    CList<CFunction*, CFunction*> CEnergize_M_Function_List;

```



```

        CList<CEnergize_M_Template*,          CEnergize_M_Template*>
CEnergize_M_Template_List;

        CList<CFunction*, CFunction*> CDistribute_E_Function_List;
        CList<CDistribute_E_Template*,      CDistribute_E_Template*>
CDistribute_E_Template_List;

        CList<CFunction*, CFunction*> CDeEn_M_Function_List;
        CList<CDeEn_M_Template*,          CDeEn_M_Template*>
CDeEn_M_Template_List;
};

```

```

#pragma once

#include "afxmt.h"
#include "geometry.h"

#define SELECTION_RADIUS 20

class CConModView :
    public CView, public CGeometry
{
protected: // create from serialization only
    CConModView();
    DECLARE_DYNCREATE(CConModView)

// Attributes
public:
    CConModDoc* GetDocument() const;

// Operations
public:

// Overrides
public:
    virtual void OnDraw(CDC* pDC);
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);

// Implementation
public:
    virtual ~CConModView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
}

```

```

protected:

// Generated message map functions
protected:
    DECLARE_MESSAGE_MAP()

//=====
//=====
//=====
// END OF WIZARD-GENERATED CODE
//=====
//=====
//=====

//=====
// SELECTION OF REASONING OPTIONS - THREE MAIN TYPES
//=====

public:
    int ReasoningOption;
        enum{QUALITATIVE_CONSERVATION,
                QUALITATIVE_IRREVERSIBILITY,
                QUANTITATIVE_EFFICIENCY,
                QUANTITATIVE_POWERREQUIRED};

//=====
// SELECTION OF MESSAGE HANDLER FUNCTIONS THROUGH ENUMERATED
WHAT-TO-DO LIST
//=====

public:
    int WhatToDo;
    enum
    {
        ESCAPE,
        ADD_FUNCTION,
        ADD_MATERIAL,
        ADD_ENERGY,
        ADD_SIGNAL,
        ADD_ENV,
        ADD_CONVERT_E_TEMPLATE,
        ADD_CONDUCT_E_TEMPLATE,           // Add more todo items here
        ADD_ENERGIZE_M_TEMPLATE,         // Add more todo items
here
        ADD_DISTRIBUTE_E_TEMPLATE,       // Add more todo items
here

```

```

        ADD_DEEN_M_TEMPLATE           // Add more todo items here
    };

    //=====
=====
    // MAIN MENU - REASONING OPTION SELECTION MESSAGE HANDLER
    FUNCTION
    //=====
=====

public:
    afx_msg void OnQualitativeConservation();
    afx_msg void OnQualitativeIrreversibility();
    afx_msg void OnQuantitativeEfficiency();
    afx_msg void OnQuantitativePowerRequired();

    //=====
=====

    // PRIMITIVES TOOLBAR MESSAGE HANDLER FUNCTIONS
    //=====
=====

public:
    void Handler_SaveFile(void);

public:
    void Handler_AddFunction(void);
    void Handler_AddMaterial(void);
    void Handler_AddEnergy(void);
    void Handler_AddSignal(void);
    void Handler_AddEnv(void);
    void Handler_EditCut(void);

    //=====
=====

    // FEATURES TOOLBAR MESSAGE HANDLER FUNCTIONS
    //=====
=====

public:
    void Handler_AddConvert_E_Template(void);
    void Handler_AddConduct_E_Template(void);
    void Handler_AddEnergize_M_Template(void);
    void Handler_AddDistribute_E_Template(void);
    void Handler_AddDeEn_M_Template(void);

    //=====
=====

    // REASONING TOOLBAR MESSAGE HANDLER FUNCTIONS
    //=====
=====

public:

```

```

void Handler_Qualitative(void);
void Handler_Quantitative(void);
//=====
=====
// FUNCTIONS FOR ADDING INSTANCES TO THE MODEL
//=====
=====
public:
    int Counter_F;
    int Counter_Env;
    int Counter_M;
    int Counter_E;
    int Counter_S;
    CString CounterString;

public:
    void AddFunction(void);
    void AddMaterial(void);
    void AddEdge_Dynamic(void);
    void AddEnergy(void);
    void AddSignal(void);
    void AddEnv(void);

public:
    void AddConvert_E_Template(void);
    void AddConduct_E_Template(void);
    void AddEnergize_M_Template(void);
    void AddDistribute_E_Template(void);
    void AddDeEn_M_Template(void);

    // The following four members are used during construction of the
dynamic
// instance of edges, and to pass their values to the final
instance.
    CElement* pTailElemDynamic;
    CElement* pHeadElemDynamic;
    bool TailNodeSelected;

//=====
=====
// FUNCTIONS FOR SELECTING INSTANCES FROM THE MODEL TO DO EDIT
OPERATIONS
//=====
=====
public:
    void Preselect(CPoint* pMouseTip);           // Preselection of
elements by mouse hover
    void Highlight(CElement* pElement);         // Change color when
preselected
    void UnHighlight(CElement* pElement);       // Reset color when
released from preselection

```

```

void SelectElement(CElement* pElement); // Finally select one
element from the presel list
void ScrollThroughPreselection(); // Scrolling through
preselected elements
POSITION ScrollPosition; // Current
position within PreselectionList that is selected
enum {NONE, TAIL, CENTER, HEAD}; // Grab handle
locations
CElement* pElementToBeDeleted;
CElement* pSelectedElement; // Pointer to
store the currently selected element

//=====
=====
// FUNCTIONS FOR STORING OBJECT POINTERS AND DETERMINING THEIR
TYPE
//=====
=====

// For basic elements (Layer 1) - Node, Edge, Function, Env, M,
E, and S

bool ElementIsNode(CElement* pElement); // TRUE if
pSelectedElement is a member of CNodeList
bool ElementIsFunction(CElement* pElement); //
TRUE if pSelectedElement is a member of CFunctionList
bool ElementIsEnv(CElement* pElement); // TRUE if
pSelectedElement is a member of CEnvList
bool ElementIsEdge(CElement* pElement); // TRUE if
pSelectedElement is a member of CEdgeList
bool ElementIsMaterial(CElement* pElement); //
TRUE if pSelectedElement is a member of CEdgeList
bool ElementIsEnergy(CElement* pElement); // TRUE if
pSelectedElement is a member of CEdgeList
bool ElementIsSignal(CElement* pElement); // TRUE if
pSelectedElement is a member of CEdgeList

POSITION NodeIndexInNodeList; // Gets set by
SelectedElementIsNode so that it could be removed
POSITION FunctionIndexInFunctionList; // Gets set by
SelectedElementIsFunction so that it could be removed
POSITION EnvIndexInEnvList; // Gets set by SelectedElementIsEnv
so that it could be removed
POSITION EdgeIndexInEdgeList; // Gets set by
SelectedElementIsEdge so that it could be removed
POSITION MaterialIndexInMaterialList; // Gets set
by SelectedElementIsEdge so that it could be removed
POSITION EnergyIndexInEnergyList; // Gets set by
SelectedElementIsEdge so that it could be removed
POSITION SignalIndexInSignalList; // Gets set by
SelectedElementIsEdge so that it could be removed

// For templates

```

```

bool ElementIsConvert_E_Function(CElement* pElement);
bool ElementIsConvert_E_Template(CElement* pElement);

bool ElementIsConduct_E_Function(CElement* pElement);
bool ElementIsConduct_E_Template(CElement* pElement);

bool ElementIsEnergize_M_Function(CElement* pElement);
bool ElementIsEnergize_M_Template(CElement* pElement);

bool ElementIsDistribute_E_Function(CElement* pElement);
bool ElementIsDistribute_E_Template(CElement* pElement);

bool ElementIsDeEn_M_Function(CElement* pElement);
bool ElementIsDeEn_M_Template(CElement* pElement);

POSITION Convert_E_Function_IndexInConvert_E_Function_List;
POSITION Convert_E_Template_IndexInConvert_E_Template_List;

POSITION Conduct_E_Function_IndexInConduct_E_Function_List;
POSITION Conduct_E_Template_IndexInConduct_E_Template_List;

POSITION Energize_M_Function_IndexInEnergize_M_Function_List;
POSITION Energize_M_Template_IndexInEnergize_M_Template_List;

POSITION Distribute_E_Function_IndexInDistribute_E_Function_List;
POSITION Distribute_E_Template_IndexInDistribute_E_Template_List;

POSITION DeEn_M_Function_IndexInDeEn_M_Function_List;
POSITION DeEn_M_Template_IndexInDeEn_M_Template_List;

void EmptyAllTempLists();
//=====
=====
// FUNCTIONS FOR EDIT OPERATIONS ON INSTANCES WITHIN THE MODEL
//=====
=====
public:
void MoveConnectDynamic();
void MoveConnect();
void DetachEdgesFromElement(CElement* pElement);
void DeleteElement(CElement* pElement);

```

```

// The following four members stores the topology of a flow
terminal
// (head or tail) that is moved by the MoveConnectDynamic
function to a temp
// storage, so that the point can be reassigned in the case the
operation
// was illegal. The storage code is in the MoveConnectDynamic
function.
// The reassignment code is in OnDraw (during gramamr chekcs).
CElement* pRememberHeadElement;
CPoint RememberHeadPoint;
CElement* pRememberTailElement;
CPoint RememberTailPoint;

//=====
// PARAMETERS AND FLAGS FOR CONTROLLING AND SIGNALLING MOUSE
POINTS AND BUTTONS
//=====
=====

public:
// Parameters
CPoint MouseLDownPoint;
CPoint MouseLUpPoint;
CPoint MouseRDownPoint;
CPoint MouseRUpPoint;
CPoint MouseMovePoint;

// Flags
bool LButtonIsDown;
bool RButtonIsDown;

//=====
=====
// MESSAGE HANDLING FUNCTIONS FOR MOUSE EVENTS
//=====
=====

public: // Mouse Button and Move Functions
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnMButtonUp(UINT nFlags, CPoint point);

public:
afx_msg BOOL OnEraseBkgnd(CDC* pDC); // Flicker elimination
afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);

//=====
=====

```

```

// CONSERVATION CHECKING FUNCTIONS - TOPOLOGICAL CONSERVATION
(WITHOUT VOCAB)
// REFERE TO: ICED-2011 PAPER
//=====
=====

bool GrammarCheckRequired;

CString Msg_OrphanFlow;
CString Msg_BarrenFlow;
CString Msg_OneInManyOut_M;
CString Msg_OneInManyOut_E;
CString Msg_ManyInOneOut_M;
CString Msg_ManyInOneOut_E;
CString Msg_ManyInManyOut;
CString Msg_MissingResidualEnergy;
CString Msg_MaterialChangeWithoutEnergy;

void Set_OrphanFlowMsg();
void Set_BarrenFlowMsg();
void Set_OneInManyOutMsg_M();
void Set_OneInManyOutMsg_E();
void Set_ManyInOneOutMsg_M();
void Set_ManyInOneOutMsg_E();
void Set_ManyInManyOutMsg();
void Set_MissingResidualEnergyMsg();
void Set_MaterialChangeWithoutEnergyMsg();

void ComposeQualitativeMessage();

//=====
=====

// Quantitative Reasoning Methods
//=====
=====

void VerifyPositivePowerOfFlows();
void VerifyEnergyBalanceOfFunctions();
void ComputeEfficiency();
void ComposeQuantitativeMessage();
bool ContinueReasoning;
};

#ifdef _DEBUG // debug version in ConModView.cpp
inline CConModDoc* CConModView::GetDocument() const
{ return reinterpret_cast<CConModDoc*>(m_pDocument); }
#endif

```



```

#pragma once
#include "Template.h"

// CConvert_E dialog

class CConvert_E_Template : public CDialog, public CTemplate
{
    DECLARE_DYNAMIC(CConvert_E_Template)

public:
    CConvert_E_Template(CWnd* pParent = NULL, CPoint InsertionPoint =
(500,500),
        CString* pCounterString_F = NULL, CString*
pCounterString_InE = NULL,
        CString* pCounterString_OutE = NULL, CString*
pCounterString_OutE_Res = NULL);

    virtual ~CConvert_E_Template();

// Dialog Data
    enum { IDD = IDD_CONVERT_E_TEMPLATE };

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
support

    DECLARE_MESSAGE_MAP()

public: // Instances that comprise the Convert_E template
    CFunction* pFunctionBlock;
    CEnergy* pEnergy_InE;
    CEnergy* pEnergy_OutE;
    CEnergy* pEnergy_OutE_Res;
};

```

```

#pragma once
#include "Template.h"

// CDeEn_M_Template dialog

class CDeEn_M_Template : public CDialog, public CTemplate
{
    DECLARE_DYNAMIC(CDeEn_M_Template)

public:
    CDeEn_M_Template(CWnd* pParent = NULL,
        CPoint InsertionPoint = (500,500),

```

```

        CString* pCounterString_F = NULL,
        CString* pCounterString_InM = NULL,
        CString* pCounterString_OutM = NULL,
        CString* pCounterString_InE = NULL,
        CString* pCounterString_OutE = NULL); // standard
constructor

        virtual ~CDeEn_M_Template();

// Dialog Data
        enum { IDD = IDD_DEEN_M_TEMPLATE };

protected:
        virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
support

        DECLARE_MESSAGE_MAP()

public: // Instances that comprise the Convert_E template
        CFunction* pFunctionBlock;
        CEnergy* pEnergy_InE;
        CEnergy* pEnergy_OutE;
        CMaterial* pMaterial_InM;
        CMaterial* pMaterial_OutM;
};

```

```

#pragma once
#include "Template.h"

// CDistribute_E_Template dialog

class CDistribute_E_Template : public CDialog, public CTemplate
{
        DECLARE_DYNAMIC(CDistribute_E_Template)

public:
        CDistribute_E_Template(CWnd* pParent = NULL,
                CPoint InsertionPoint = (500,500),
                CString* pCounterString_F = NULL,
                CString* pCounterString_InE = NULL,
                CString* pCounterString_OutE1 = NULL,
                CString* pCounterString_OutE2 = NULL); // standard
constructor

        virtual ~CDistribute_E_Template();

// Dialog Data
        enum { IDD = IDD_DISTRIBUTE_E_TEMPLATE };

protected:

```

```

        virtual void DoDataExchange(CDataExchange* pDX);      // DDX/DDV
support

        DECLARE_MESSAGE_MAP()

public:          // Instances that comprise the Convert_E template
        CFunction* pFunctionBlock;
        CEnergy* pEnergy_InE;
        CEnergy* pEnergy_OutE1;
        CEnergy* pEnergy_OutE2;
};

```

```

#pragma once
#include "element.h"
#include "math.h"
#include "node.h"

#define EDGE_HEAD_SIZE 20
#define EDGE_HEAD_HALF_ANGLE 0.25 // Radians

class CEdge : public CElement
{
public:
    CEdge(void);
    CEdge(CPoint TailClick, CPoint HeadClick);
    ~CEdge(void);

    // Head construction data
    CPoint HeadLeftVertex, HeadRightVertex;
    double HeadSize, HalfHeadAngle;
    CPoint HeadVertexArray[3];

    // Topological information
    void ComputeAnchorPoints();
    void AttachEdgeToNearestAnchor();
    void ResetGeometricCenter(); // Makes sure that the
    GeometricCenter is reset between the // Tail and Head
    points, when an arrow is moved by grabbing // Those terminal
    points
    bool ThisFlowIsIncomingBaggage;
    bool ThisFlowIsOutgoingBaggage;

    // Drawing data
    int StemThickness;
    int StemLineFont;
    enum {NONE, THIN, MEDIUM, THICK};
    int FontSize;

```

```
void DrawOnDC(CDC* pDC);  
};
```

```
#pragma once  
#include "geometry.h"  
  
#define GENERIC_PEN_R 0  
#define GENERIC_PEN_G 0  
#define GENERIC_PEN_B 0  
  
#define GENERIC_BRUSH_R 0  
#define GENERIC_BRUSH_G 0  
#define GENERIC_BRUSH_B 0  
  
#define DANGLING_BRUSH_R 255  
#define DANGLING_BRUSH_G 0  
#define DANGLING_BRUSH_B 0  
  
#define PRESELECTION_PEN_R 200  
#define PRESELECTION_PEN_G 0  
#define PRESELECTION_PEN_B 200  
  
#define SELECTION_PEN_R 0  
#define SELECTION_PEN_G 200  
#define SELECTION_PEN_B 0  
  
#define RESIDUAL_PEN_R 255  
#define RESIDUAL_PEN_G 0  
#define RESIDUAL_PEN_B 0  
  
#define GENERIC_FONT_SIZE 16  
#define BAGGAGE_FONT_SIZE 12  
  
class CElement :  
    public CGeometry/*, public CDialog*/  
{  
public:  
    CElement(void);  
    virtual ~CElement(void);           // Must be virtual, so that  
    individual desctrutors of          // the derived classes  
    are called when CConModView's     // destructor tries to  
    close the session  
  
    // PARAMETERS OVERRIDEN IN BOTH CNode AND CEdge CLASSES  
    bool IsHighlighted;  
    bool IsSelected;  
    bool IsResidual;
```

```

    CPoint GeometricCenter;
    CPoint Anchors[16];
    CPoint AnchorsForBaggageFlows[16];

    //CString GivenName;          // Unnecessary - the individual
classes need their own          // GivenName attribute,
because the dilaog constructor  // needs a GivenName
that is not inherited.

    int PenR, PenG, PenB;
    int BrushR, BrushG, BrushB;

    int GrabHandle;          // Stores where (Head, Tail, Center) an
element is grabbed by the mouse
    virtual void DrawOnDC(CDC* pDC);

    //int ReasoningOption;
        enum{QUALITATIVE_CONSERVATION,
                QUALITATIVE_IRREVERSIBILITY,
                QUANTITATIVE_EFFICIENCY,
                QUANTITATIVE_POWERREQUIRED};

    // PARAMETERS OVERRIDEN IN CEDGE: TOPOLOGY DATA
    CPoint TailPoint, HeadPoint;
    int HeadBrushR, HeadBrushG, HeadBrushB;
    int TailBrushR, TailBrushG, TailBrushB;
    CElement* pHeadElem;
    CElement* pTailElem;
};

```

```

#pragma once
#include "Template.h"

// CEnergize_M_Template dialog

class CEnergize_M_Template : public CDialog, public CTemplate
{
    DECLARE_DYNAMIC(CEnergize_M_Template)

public:
    CEnergize_M_Template(CWnd* pParent = NULL,
        CPoint InsertionPoint = (500,500),
        CString* pCounterString_F = NULL,
        CString* pCounterString_InM = NULL,

```

```

        CString* pCounterString_OutM = NULL,
        CString* pCounterString_InE = NULL,
        CString* pCounterString_OutE = NULL);
    virtual ~CEnergize_M_Template();

// Dialog Data
    enum { IDD = IDD_ENERGIZE_M_TEMPLATE };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

    DECLARE_MESSAGE_MAP()

public:    // Instances that comprise the Convert_E template
    CFunction* pFunctionBlock;
    CEnergy* pEnergy_InE;
    CEnergy* pEnergy_OutE;
    CMaterial* pMaterial_InM;
    CMaterial* pMaterial_OutM;
};

```

```

#pragma once
#include "edge.h"
#include "afxcmn.h"
#include "afxwin.h"

// CEnergy dialog

class CEnergy :
    public CEdge, public CDialog
{
    DECLARE_DYNAMIC(CEnergy)

public:
    CEnergy(CWnd* pParent = NULL,
            CPoint TailClick = (0,0,0),
            CPoint HeadClick = (100,100,0),
            CString* pCounterString = NULL,
            int ReasOpt = QUALITATIVE_CONSERVATION);    // standard
constructor

    virtual ~CEnergy();

// Dialog Data
    enum { IDD = IDD_ENERGY };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

```

```

        DECLARE_MESSAGE_MAP()
public:

    CList<CEnergy*, CEnergy*> ChildList;
    CList<CEnergy*, CEnergy*> ParentList;
    CString GivenName;
    void DrawOnDC(CDC*pDC);
    int UI_IsResidual;
    BOOL OnInitDialog();
    void OnOK();
    CTreeCtrl* pEnergyTaxonomy;
    HTREEITEM hEnergyType;
    CString EnergyTypeName;

    // Quantitative data members
    double Power;
    double UI_ForceTerm, UI_RateTerm;
    int ReasoningOption;

};

```

```

#pragma once
#include "node.h"

#define ENV_SIZE 25

#define ENV_BRUSH_R 255
#define ENV_BRUSH_G 220
#define ENV_BRUSH_B 210

// CEnv dialog

class CEnv :
    public CNode, public CDialog
{
    DECLARE_DYNAMIC(CEnv)

public:
    CEnv(CWnd* pParent = NULL, CPoint InsertionPoint = (500,500,0),
        CString* pCounterString = NULL); // standard constructor
    virtual ~CEnv();

    // Dialog Data
    enum { IDD = IDD_ENV };

protected:

```

```

        virtual void DoDataExchange(CDataExchange* pDX);        // DDX/DDV
support

        DECLARE_MESSAGE_MAP()
public:
        // Environment name within block
        CString GivenName;

        // Drawing functions
        void ComputeBlockCoordinates();
        void DrawOnDC(CDC* pDC);
};

```

```

#pragma once
#include "node.h"

#define BLOCK_LENGTH 80
#define BLOCK_HEIGHT 40

#define FUNCTION_BRUSH_R 150
#define FUNCTION_BRUSH_G 175
#define FUNCTION_BRUSH_B 200

// CFunction dialog

class CFunction :
    public CNode, public CRect, public CDialog
{
    DECLARE_DYNAMIC(CFunction)

public:
    CFunction(CWnd* pParent = NULL, CPoint InsertionPoint =
(500,500,0), CString* pCounterString = NULL); // standard constructor
    virtual ~CFunction();

// Dialog Data
    enum {IDD = IDD_FUNCTION};

protected:
    virtual void DoDataExchange(CDataExchange* pDX);        // DDX/DDV
support

    DECLARE_MESSAGE_MAP()
public:

        // Function name within block
        CString GivenName;

        // Drawing functions
        void ComputeBlockCoordinates();

```



```
void DrawOnDC(CDC* pDC);

// Quantitative data
double Efficiency;
};
```

```
#pragma once
#include "math.h"

class CGeometry
{
public:
    CGeometry(void);
    ~CGeometry(void);

    // Member Functions
    int RoundToInteger(long n, int t);
    CPoint SnapToGrid(CPoint p);
    long distance(CPoint p1, CPoint p2);
    CPoint* InterpolatePoints(CPoint p1, CPoint p2, double ratio);
};
```

```
// MainFrm.h : interface of the CMainFrame class
//

#pragma once

class CMainFrame : public CMDIFrameWnd
{
    DECLARE_DYNAMIC(CMainFrame)
public:
    CMainFrame();

// Attributes
public:

// Operations
public:

// Overrides
public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected: // control bar embedded members
    CStatusBar m_wndStatusBar;
    CToolBar m_wndToolBar;
    // Conmod custom toolbars
    CToolBar m_primitivesToolBar;
    CToolBar m_featuresToolBar;
    CToolBar m_reasoningToolBar;

// Generated message map functions
protected:
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    DECLARE_MESSAGE_MAP()
};

```

```

#pragma once
#include "edge.h"

// CMaterial dialog

class CMaterial :
    public CEdge, public CDialog
{
    DECLARE_DYNAMIC(CMaterial)

public:
    CMaterial(CWnd* pParent = NULL,
        CPoint TailClick = (0,0,0),
        CPoint HeadClick = (100,100,0),
        CString* pCounterString = NULL,
        int ReasOpt = QUALITATIVE_CONSERVATION); // standard

    constructor
    virtual ~CMaterial();

// Dialog Data
    enum { IDD = IDD_MATERIAL };

protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV
support

    DECLARE_MESSAGE_MAP()

```

```

public:
    CList<CMaterial*, CMaterial*> ChildList;
    CList<CMaterial*, CMaterial*> ParentList;
    CString GivenName;
    void DrawOnDC(CDC*pDC);
    int UI_IsResidual;
    BOOL OnInitDialog();
    void OnOK();
    CTreeCtrl* pMaterialTaxonomy;
    HTREEITEM hMaterialType;
    CString MaterialTypeName;
    int ReasoningOption;
};

```

```

#pragma once
#include "afxwin.h"
#ifndef _MEMDC_H_
#define _MEMDC_H_

////////////////////////////////////

// CMemDC - memory DC
//
// Author: Keith Rule
// Email: keithr@europa.com
// Copyright 1996-2002, Keith Rule
//
// You may freely use or modify this code provided this
// Copyright is included in all derived versions.
//
// History - 10/3/97 Fixed scrolling bug.
//           Added print support. - KR
//
//           11/3/99 Fixed most common complaint. Added
//           background color fill. - KR
//
//           11/3/99 Added support for mapping modes other than
//           MM_TEXT as suggested by Lee Sang Hun. - KR
//
//           02/11/02 Added support for CScrollView as supplied
//           by Gary Kirkham. - KR
//
// This class implements a memory Device Context which allows
// flicker free drawing.

class CMemDC : public CDC {
private:
    CBitmap      m_bitmap;           // Offscreen bitmap
    CBitmap*     m_oldBitmap;       // bitmap originally found in CMemDC
    CDC*         m_pDC;             // Saves CDC passed in constructor

```

```

    CRect      m_rect;          // Rectangle of drawing area.
    BOOL      m_bMemDC;        // TRUE if CDC really is a Memory DC.

public:
    CMemDC(CDC* pDC, const CRect* pRect = NULL) : CDC()
    {
        ASSERT(pDC != NULL);

        // Some initialization

        m_pDC = pDC;
        m_oldBitmap = NULL;
        m_bMemDC = !pDC->IsPrinting();

        // Get the rectangle to draw

        if (pRect == NULL) {
            pDC->GetClipBox(&m_rect);
        } else {
            m_rect = *pRect;
        }

        if (m_bMemDC) {
            // Create a Memory DC

            CreateCompatibleDC(pDC);
            pDC->LPtoDP(&m_rect);

            m_bitmap.CreateCompatibleBitmap(pDC, m_rect.Width(),
                                           m_rect.Height());
            m_oldBitmap = SelectObject(&m_bitmap);

            SetMapMode(pDC->GetMapMode());

            SetWindowExt(pDC->GetWindowExt());
            SetViewportExt(pDC->GetViewportExt());

            pDC->DPtoLP(&m_rect);
            SetWindowOrg(m_rect.left, m_rect.top);
        } else {
            // Make a copy of the relevent parts of the current

            // DC for printing

            m_bPrinting = pDC->m_bPrinting;
            m_hDC        = pDC->m_hDC;
            m_hAttribDC  = pDC->m_hAttribDC;
        }

        // Fill background

        FillSolidRect(m_rect, pDC->GetBkColor());
    }

```

```

~CMemDC()
{
    if (m_bMemDC) {
        // Copy the offscreen bitmap onto the screen.

        m_pDC->BitBlt(m_rect.left, m_rect.top,
                    m_rect.Width(), m_rect.Height(),
                    this, m_rect.left, m_rect.top, SRCCOPY);

        //Swap back the original bitmap.

        SelectObject(m_oldBitmap);
    } else {
        // All we need to do is replace the DC with an illegal
        // value, this keeps us from accidentally deleting the
        // handles associated with the CDC that was passed to
        // the constructor.

        m_hDC = m_hAttribDC = NULL;
    }
}

// Allow usage as a pointer

CMemDC* operator->()
{
    return this;
}

// Allow usage as a pointer

operator CMemDC*()
{
    return this;
}
};

#endif

```

```

#pragma once
#include "element.h"

class CNode :
    public CElement
{
public:

```

```

    CNode(void);
    virtual ~CNode(void); // Must be virtual, so that individual
desctrutors of // the derived classes are
called when CConModView's // destructor tries to close
the session

    // Parameters to check for dangling functions and env instances
    bool NoInputAttached;
    bool NoOutputAttached;

    void ComputeBlockCoordinates();
};

```

```

//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by ConMod.rc
//
#define IDD_ABOUTBOX 100
#define IDP_OLE_INIT_FAILED 100
#define IDS_STRING101 101
#define IDS_STRING102 102
#define IDD_CONVERT_E_TEMPLATE 104
#define IDD_Conduct_E_TEMPLATE 105
#define IDD_ENERGIZE_M_TEMPLATE 106
#define IDD_DISTRIBUTE_E_TEMPLATE 107
#define IDD_DEEN_M_TEMPLATE 108
#define IDR_MAINFRAME 128
#define IDR_ConceptTYPE 129
#define IDC_CROSSHAIR 130
#define IDR_PRIMITIVES 131
#define IDD_FUNCTION 138
#define IDD_MATERIAL 139
#define IDD_ENERGY 140
#define IDD_SIGNAL 141
#define IDD_ENV 142
#define IDR_FEATURES 143
#define IDR_REASONING 145
#define IDR_HTML1 147
#define IDR_HTML2 148
#define IDC_FUNCTION_NAME 1006
#define IDC_MATERIAL_NAME 1010
#define IDC_ENERGY_NAME 1011
#define IDC_SIGNAL_NAME 1012
#define IDC_ENV_NAME 1013
#define IDC_RESIDUAL_ENERGY 1014
#define IDC_RESIDUAL_MATERIAL 1021
#define IDC_FORCE_TERM 1023
#define IDC_RATE_TERM 1024

```

```

#define IDC_STATIC1 1025
#define IDC_STATIC2 1026
#define IDC_FORCE_STATIC_TEXT 1031
#define IDC_FORCE_STATIC_TEXT2 1032
#define IDC_RATE_STATIC_TEXT 1032
#define ID_ADD_FUNCTION 32779
#define ID_ADD_SIGNAL 32782
#define ID_ADD_ENERGY 32783
#define ID_ADD_MATERIAL 32784
#define ID_ADD_ENV 32785
#define ID_IMPORT 32786
#define ID_CONVERT_E 32787
#define ID_BUTTON32789 32789
#define ID_CONDUCT_E_TEMPLATE 32789
#define ID_ENERGIZE_M_TEMPLATE 32790
#define ID_DISTRIBUTE_E_TEMPLATE 32791
#define ID_BUTTON32792 32792
#define ID_DEEN_M_TEMPLATE 32792
#define ID_BUTTON32794 32794
#define ID_QUANTITATIVE 32795
#define ID_REASONINGOPTION_QUALITATIVE 32796
#define ID_REASONINGOPTION_POWERREQUIRED 32798
#define ID_QUALITATIVE 32799
#define ID_HELP_CONMODDOCUMENTATION 32801
#define ID_Menu 32802
#define ID_BUTTON32805 32805
#define ID_QUANTITATIVE_EFFICIENCY 32807
#define ID_QUANTITATIVE_POWERREQUIRED 32808
#define ID_QUALITATIVE_CONSERVATIN 32809
#define ID_QUALITATIVE_IRREVERSIBILITY 32810
#define ID_QUALITATIVE_CONSERVATION 32811
#define ID_QUALITATIVE_CONSERVATI 32812

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 149
#define _APS_NEXT_COMMAND_VALUE 32813
#define _APS_NEXT_CONTROL_VALUE 1032
#define _APS_NEXT_SYMED_VALUE 109
#endif
#endif
#endif

```

```

#pragma once
#include "edge.h"

// CSignal dialog

```

```

class CSignal :
    public CEdge, public CDialog
{
    DECLARE_DYNAMIC(CSignal)

public:
    CSignal(CWnd* pParent = NULL, CPoint TailClick = (0,0,0), CPoint
HeadClick = (100,100,0), CString* pCounterString = NULL);    // standard
constructor
    virtual ~CSignal();

// Dialog Data
    enum { IDD = IDD_SIGNAL };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

    DECLARE_MESSAGE_MAP()

public:
    //CList<CSignal*, CSignal*> ChildList;
    //CList<CSignal*, CSignal*> ParentList;
    CString GivenName;
    void DrawOnDC(CDC*pDC);
};

```

```

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently,
// but are changed infrequently

#pragma once

#ifdef _SECURE_ATL
#define _SECURE_ATL 1
#endif

#ifdef VC_EXTRALEAN
#define VC_EXTRALEAN    // Exclude rarely-used stuff from
Windows headers
#endif

#include "targetver.h"
#include "memdc.h"    // Flicker prevention

#define _ATL_CSTRING_EXPLICIT_CONSTRUCTORS    // some CString
constructors will be explicit

// turns off MFC's hiding of some common and often safely ignored
warning messages

```



```

#define _AFX_ALL_WARNINGS

#include <afxwin.h>           // MFC core and standard components
#include <afxext.h>           // MFC extensions

#include <afxdisp.h>         // MFC Automation classes

#ifndef _AFX_NO_OLE_SUPPORT
#include <afxdtctl.h>        // MFC support for Internet Explorer 4
Common Controls
#endif
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>          // MFC support for Windows
Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT

#ifdef _UNICODE
#if defined _M_IX86
#pragma comment(linker, "/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls'          version='6.0.0.0'
processorArchitecture='x86'                        publicKeyToken='6595b64144ccf1df'
language='*'\")
#elif defined _M_IA64
#pragma comment(linker, "/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls'          version='6.0.0.0'
processorArchitecture='ia64'                       publicKeyToken='6595b64144ccf1df'
language='*'\")
#elif defined _M_X64
#pragma comment(linker, "/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls'          version='6.0.0.0'
processorArchitecture='amd64'                      publicKeyToken='6595b64144ccf1df'
language='*'\")
#else
#pragma comment(linker, "/manifestdependency:\"type='win32'
name='Microsoft.Windows.Common-Controls'          version='6.0.0.0'
processorArchitecture='*'                           publicKeyToken='6595b64144ccf1df'
language='*'\")
#endif
#endif

```

```

#pragma once

// The following macros define the minimum required platform. The
// minimum required platform
// is the earliest version of Windows, Internet Explorer etc. that has
// the necessary features to run
// your application. The macros work by enabling all features
// available on platform versions up to and

```

```

// including the version specified.

// Modify the following defines if you have to target a platform prior
// to the ones specified below.
// Refer to MSDN for the latest info on corresponding values for
// different platforms.
#ifndef WINVER // Specifies that the minimum
required platform is Windows Vista.
#define WINVER 0x0600 // Change this to the appropriate value
to target other versions of Windows.
#endif

#ifndef _WIN32_WINNT // Specifies that the minimum required
platform is Windows Vista.
#define _WIN32_WINNT 0x0600 // Change this to the appropriate value
to target other versions of Windows.
#endif

#ifndef _WIN32_WINDOWS // Specifies that the minimum required
platform is Windows 98.
#define _WIN32_WINDOWS 0x0410 // Change this to the appropriate value
to target Windows Me or later.
#endif

#ifndef _WIN32_IE // Specifies that the minimum
required platform is Internet Explorer 7.0.
#define _WIN32_IE 0x0700 // Change this to the appropriate value
to target other versions of IE.
#endif

```

```

#pragma once

#include "Element.h"
#include "Function.h"
#include "Env.h"
#include "Material.h"
#include "Energy.h"
#include "Signal.h"

#define TEMPLATE_FLOW_LENGTH 120

// CTemplate

class CTemplate : public CElement
{
public:
    CTemplate();
    virtual ~CTemplate();

```

```

};

/*
    This is a high-level abstract class for all templates of the
    second layer.
    The purpose is to provide one identity so that instances all
    Layer-2 versb, such as
    Covnert_E and Energize_M, can be stored in a single list called
    CTemplateList,
    declared in the Doc class as usual. The template instances are
    not used in the
    model in their own identity, they are only required to
    instnatiat the elements
    such as functions and flows WITHIN the templates using one
    instance call in
    View, such as in functions AddCovnert_E. After that, the
    elements are used, while
    the template instance must be deleted. To facilitate this
    delete, the templates
    are stored in this CTemplateList, which is emptied during
    application exit (View class
    desctrictor).
*/

```

## Appendix D. Source Files for the ConMod Application

```
// ChildFrm.cpp : implementation of the CChildFrame class
//
#include "stdafx.h"
#include "ConMod.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
END_MESSAGE_MAP()

// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
    // TODO: add member initialization code here
}

CChildFrame::~CChildFrame()
{
}

BOOL CChildFrame::OnCreateClient(LPCREATESTRUCT /*lpcs*/,
CCreateContext* pContext)
{
    return m_wndSplitter.Create(this,
        2, 2, // TODO: adjust the number of rows,
        columns
        CSize(10, 10), // TODO: adjust the minimum pane size
        pContext);
}

BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying the
    CREATESTRUCT cs
    if( !CMDIChildWnd::PreCreateWindow(cs) )
        return FALSE;

    cs.style |= WS_MAXIMIZE; // This does not maximize the child
    window without the next line
}
```

```

        cs.style |= WS_VISIBLE; // These pipe characters (|) are very
important - try deleting them!
        return TRUE;
    }

// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}

#endif // _DEBUG

// CChildFrame message handlers

```

```

// Conduct_E_Template.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Conduct_E_Template.h"

// CConduct_E_Template dialog

IMPLEMENT_DYNAMIC(CConduct_E_Template, CDialog)

CConduct_E_Template::CConduct_E_Template(CWnd* pParent /*= NULL*/,
CPoint InsertionPoint /*= (500,500)*/,
        CString* pCounterString_F /*= NULL*/,
CString* pCounterString_InE /*= NULL*/,
        CString* pCounterString_OutE /*=
NULL*/, CString* pCounterString_OutE_Res /*= NULL*/)
    : CDialog(CConduct_E_Template::IDD, pParent)
{
    pFunctionBlock = new CFunction(NULL, InsertionPoint,
pCounterString_F);

    CPoint TailOfInE(InsertionPoint.x - TEMPLATE_FLOW_LENGTH,
InsertionPoint.y);

```

```

        CPoint  HeadOfOutE(InsertionPoint.x  +  TEMPLATE_FLOW_LENGTH,
InsertionPoint.y);
        CPoint  HeadOfOutE_Res(InsertionPoint.x,  InsertionPoint.y  +
TEMPLATE_FLOW_LENGTH);

        pEnergy_InE  =  new  CEnergy(NULL,  TailOfInE,  InsertionPoint,
pCounterString_InE);
        pEnergy_OutE  =  new  CEnergy(NULL,  InsertionPoint,  HeadOfOutE,
pCounterString_OutE);
        pEnergy_OutE_Res  =  new  CEnergy(NULL,  InsertionPoint,
HeadOfOutE_Res, pCounterString_OutE_Res);

        pEnergy_InE->pHeadElem = pFunctionBlock;
        pEnergy_OutE->pTailElem = pFunctionBlock;
        pEnergy_OutE_Res->pTailElem = pFunctionBlock;
        pEnergy_OutE_Res->UI_IsResidual = true;
    }

CConduct_E_Template::~CConduct_E_Template()
{
}

void CConduct_E_Template::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CConduct_E_Template, CDialog)
END_MESSAGE_MAP()

// CConduct_E_Template message handlers

```

```

// ConMod.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "ConMod.h"
#include "MainFrm.h"

#include "ChildFrm.h"
#include "ConModDoc.h"
#include "ConModView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

```

```

// CConModApp
BEGIN_MESSAGE_MAP(CConModApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, &CConModApp::OnAppAbout)
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, &CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, &CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, &CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// CConModApp construction

CConModApp::CConModApp()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
    EnableHtmlHelp();
}

// The one and only CConModApp object

CConModApp theApp;

// CConModApp initialization

BOOL CConModApp::InitInstance()
{
    // InitCommonControlsEx() is required on Windows XP if an
application
    // manifest specifies use of ComCtl32.dll version 6 or later to
enable
    // visual styles. Otherwise, any window creation will fail.
    INITCOMMONCONTROLSEX InitCtrls;
    InitCtrls.dwSize = sizeof(InitCtrls);
    // Set this to include all the common control classes you want to
use
    // in your application.
    InitCtrls.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&InitCtrls);

    CWinApp::InitInstance();

    // Initialize OLE libraries
    if (!AfxOleInit())
    {
        AfxMessageBox(IDP_OLE_INIT_FAILED);
        return FALSE;
    }
    AfxEnableControlContainer();
}

```

```

// Standard initialization
// If you are not using these features and wish to reduce the
size
// of your final executable, you should remove from the following
// the specific initialization routines you do not need
// Change the registry key under which our settings are stored
// TODO: You should modify this string to be something
appropriate
// such as the name of your company or organization
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings(4); // Load standard INI file options
(including MRU)
// Register the application's document templates. Document
templates
// serve as the connection between documents, frame windows and
views
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(IDR_ConceptTYPE,
    RUNTIME_CLASS(CConModDoc),
    RUNTIME_CLASS(CChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CConModView));
if (!pDocTemplate)
    return FALSE;
AddDocTemplate(pDocTemplate);

// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame || !pMainFrame->LoadFrame(IDR_MAINFRAME))
{
    delete pMainFrame;
    return FALSE;
}
m_pMainWnd = pMainFrame;
// call DragAcceptFiles only if there's a suffix
// In an MDI app, this should occur immediately after setting
m_pMainWnd
// Enable drag/drop open
m_pMainWnd->DragAcceptFiles();

// Enable DDE Execute open
EnableShellOpen();
RegisterShellFileTypes(TRUE);

// Parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

// Dispatch commands specified on the command line. Will return
FALSE if
// app was launched with /RegServer, /Register, /Unregserver or
/Unregister.
if (!ProcessShellCommand(cmdInfo))

```



```

        return FALSE;
        // The main window has been initialized, so show and update it
        pMainFrame->ShowWindow(m_nCmdShow);
        pMainFrame->UpdateWindow();

        return TRUE;
    }

// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
    CAboutDlg();

// Dialog Data
    enum { IDD = IDD_ABOUTBOX };

protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV
support

// Implementation
protected:
    DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
END_MESSAGE_MAP()

// App command to run the dialog
void CConModApp::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}

// CConModApp message handlers

```

```

// ConModDoc.cpp : implementation of the CConModDoc class
//

#include "stdafx.h"
#include "ConMod.h"

#include "ConModDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CConModDoc

IMPLEMENT_DYNCREATE(CConModDoc, CDocument)

BEGIN_MESSAGE_MAP(CConModDoc, CDocument)
END_MESSAGE_MAP()

// CConModDoc construction/destruction

CConModDoc::CConModDoc()
{
    // TODO: add one-time construction code here
}

CConModDoc::~CConModDoc()
{
}

BOOL CConModDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}

// CConModDoc serialization

void CConModDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {

```

```

        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}

// CConModDoc diagnostics

#ifdef _DEBUG
void CConModDoc::AssertValid() const
{
    CDocument::AssertValid();
}

void CConModDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif // _DEBUG

// CConModDoc commands

```

```

// ConModView.cpp : implementation of the CConModView class

#include "stdafx.h"
#include "ConMod.h"
#include "ConModDoc.h"
#include "ConModView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

//
=====
=====
//
=====
=====
// MESSAGE MAPS
//
=====
=====
//
=====
=====

```

```

IMPLEMENT_DYNCREATE(CConModView, CView)

BEGIN_MESSAGE_MAP(CConModView, CView)

    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, &CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, &CView::OnFilePrintPreview)

    // ConMod Main Menu - Reasoning Options Message Handlers
    ON_COMMAND(ID_QUALITATIVE_CONSERVATION,
&CConModView::OnQualitativeConservation)
    ON_COMMAND(ID_QUALITATIVE_IRREVERSIBILITY,
&CConModView::OnQualitativeIrreversibility)
    ON_COMMAND(ID_QUANTITATIVE EFFICIENCY,
&CConModView::OnQuantitativeEfficiency)
    ON_COMMAND(ID_QUANTITATIVE_POWERREQUIRED,
&CConModView::OnQuantitativePowerRequired)

    // ConMod PRIMITIVES Toolbar Commands
    ON_COMMAND(ID_ADD_FUNCTION, Handler_AddFunction)
    ON_COMMAND(ID_ADD_MATERIAL, Handler_AddMaterial)
    ON_COMMAND(ID_ADD_ENERGY, Handler_AddEnergy)
    ON_COMMAND(ID_ADD_SIGNAL, Handler_AddSignal)
    ON_COMMAND(ID_ADD_ENV, Handler_AddEnv)

    // ConMod FEATURES Toolbar Commands
    ON_COMMAND(ID_CONVERT_E, Handler_AddConvert_E_Template)
    ON_COMMAND(ID_CONDUCT_E_TEMPLATE, Handler_AddConduct_E_Template)
    ON_COMMAND(ID_ENERGIZE_M_TEMPLATE,
Handler_AddEnergize_M_Template)
    ON_COMMAND(ID_DISTRIBUTE_E_TEMPLATE,
Handler_AddDistribute_E_Template)
    ON_COMMAND(ID_DEEN_M_TEMPLATE, Handler_AddDeEn_M_Template)

    // ConMod REASONING Toolbar Commands
    ON_COMMAND(ID_QUALITATIVE, Handler_Qualitative)
    ON_COMMAND(ID_QUANTITATIVE, Handler_Quantitative)

    // ConMod Mouse Event Commands
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_RBUTTONDOWN()
    ON_WM_RBUTTONUP()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONDBLCLK()

    // Flicker prevention of the screen
    ON_WM_ERASEBKGND()

    // ConModKeyboard Event Commands
    ON_COMMAND(ID_EDIT_CUT, Handler_EditCut)

```

```

        ON_WM_MBUTTONDOWN()
END_MESSAGE_MAP()

//
=====
//
=====
// CONSTRUCTOR and DESTRUCTOR
//
=====
//
=====

CConModView::CConModView()
{
    ReasoningOption = QUALITATIVE_CONSERVATION;
    ContinueReasoning = true;

    WhatToDo = ESCAPE;
    LButtonIsDown = false;
    RButtonIsDown = false;
    pTailElemDynamic = NULL;
    pHeadElemDynamic = NULL;
    TailNodeSelected = false;
    pElementToBeDeleted = NULL;

    // Conservation Checking Messages
    Msg_OrphanFlow = "";
    Msg_BarrenFlow = "";
    Msg_OneInManyOut_M = "";
    Msg_OneInManyOut_E = "";
    Msg_ManyInOneOut_M = "";
    Msg_ManyInOneOut_E = "";
    Msg_ManyInManyOut = "";
    Msg_MissingResidualEnergy = "";
    Msg_MaterialChangeWithoutEnergy = "";

    Counter_F = 0;
    Counter_Env = 0;
    Counter_M = 0;
    Counter_E = 0;
    Counter_S = 0;

    GrammarCheckRequired = true;
}

CConModView::~CConModView()
{

```

```

        CConModDoc* pDoc = GetDocument();

        for (POSITION pos = pDoc->CElementList.GetHeadPosition(); pos !=
NULL; )
        {
            delete pDoc->CElementList.GetAt(pos);
            pDoc->CElementList.GetNext(pos);
        }
        pDoc->CFunctionList.RemoveAll();
        pDoc->CEnvList.RemoveAll();
        pDoc->CNodeList.RemoveAll();
        pDoc->CMaterialList.RemoveAll();
        pDoc->CEnergyList.RemoveAll();
        pDoc->CSignalList.RemoveAll();
        pDoc->CEdgeList.RemoveAll();
        pDoc->CElementList.RemoveAll();

        for (POSITION pos = pDoc->CTemplateList.GetHeadPosition(); pos !=
NULL; )
        {
            delete pDoc->CTemplateList.GetAt(pos);
            pDoc->CTemplateList.GetNext(pos);
        }
        pDoc->CTemplateList.RemoveAll();
    }

    BOOL CConModView::PreCreateWindow(CREATESTRUCT& cs)
    {
        // TODO: Modify the Window class or styles here by modifying
        // the CREATESTRUCT cs
        cs.lpszClass = AfxRegisterWndClass(CS_DBLCLKS | CS_HREDRAW |
CS_VREDRAW,
            AfxGetApp()->LoadCursor(IDC_CROSSHAIR), (HBRUSH)
(COLOR_WINDOW + 1));

        return CView::PreCreateWindow(cs);
    }

    //
    =====
    //
    =====
    // DRAWING: OnDraw FUNCTION
    //
    =====
    //
    =====

    void CConModView::OnDraw(CDC* dc)

```

```

{
    CMemDC pDC(dc);

    CConModDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    //=====
    // Check for dangling CNodes (CFunctions and CEnvs)
    //=====

    if (pDoc->CNodeList.IsEmpty() == false)
    {
        for (POSITION pos = pDoc->CNodeList.GetHeadPosition(); pos
        != NULL; )
        {
            pDoc->CNodeList.GetAt(pos)->NoInputAttached = true;
            pDoc->CNodeList.GetAt(pos)->NoOutputAttached = true;

            for (POSITION pos_inner = pDoc-
            >CEdgeList.GetHeadPosition(); pos_inner != NULL; )
            {
                if (pDoc->CEdgeList.GetAt(pos_inner)->pTailElem
                == pDoc->CNodeList.GetAt(pos))
                    pDoc->CNodeList.GetAt(pos)-
                    >NoOutputAttached = false;
                if (pDoc->CEdgeList.GetAt(pos_inner)->pHeadElem
                == pDoc->CNodeList.GetAt(pos))
                    pDoc->CNodeList.GetAt(pos)-
                    >NoInputAttached = false;
                pDoc->CEdgeList.GetNext(pos_inner);
            }

            pDoc->CNodeList.GetNext(pos);
        }
    }

    //=====
    // Check for baggage flows (incoming and outgoing)
    //=====

    if (pDoc->CEdgeList.IsEmpty() == false)
    {
        for (POSITION pos = pDoc->CEdgeList.GetHeadPosition(); pos
        != NULL; )
        {
            if (ElementIsEdge(pDoc->CEdgeList.GetAt(pos)-
            >pHeadElem) &&

```

```

        (pDoc->CEdgeList.GetAt(pos)->pHeadElem-
>pTailElem == pDoc->CEdgeList.GetAt(pos)->pTailElem) &&
        ElementIsNode(pDoc->CEdgeList.GetAt(pos)-
>pTailElem))
        pDoc->CEdgeList.GetAt(pos)-
>ThisFlowIsOutgoingBaggage = true;
        else
        pDoc->CEdgeList.GetAt(pos)-
>ThisFlowIsOutgoingBaggage = false;

        if
        (ElementIsEdge(pDoc->CEdgeList.GetAt(pos)-
>pTailElem) &&
        (pDoc->CEdgeList.GetAt(pos)->pTailElem-
>pHeadElem == pDoc->CEdgeList.GetAt(pos)->pHeadElem) &&
        ElementIsNode(pDoc->CEdgeList.GetAt(pos)-
>pHeadElem))
        pDoc->CEdgeList.GetAt(pos)-
>ThisFlowIsIncomingBaggage = true;
        else
        pDoc->CEdgeList.GetAt(pos)-
>ThisFlowIsIncomingBaggage = false;

        pDoc->CEdgeList.GetNext(pos);
    }
}

//=====
// Update the ReasoningOption variable in Energy flows, so that
dialogs
// show the correct reasoning option check box through
ONInitDialog
//=====
if (pDoc->CEnergyList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition();
pos != NULL; )
    {
        pDoc->CEnergyList.GetAt(pos)->ReasoningOption = this-
>ReasoningOption;
        pDoc->CElementList.GetNext(pos);
    }
}

//=====
// Update the ReasoningOption variable in Material flows, so that
dialogs
// show the correct reasoning option check box through
ONInitDialog
//=====
if (pDoc->CMaterialList.IsEmpty() == false)
{

```



```

        for (POSITION pos = pDoc->CMaterialList.GetHeadPosition();
pos != NULL; )
        {
            pDoc->CMaterialList.GetAt (pos) ->ReasoningOption      =
this->ReasoningOption;
            pDoc->CMaterialList.GetNext (pos);
        }
    }

    //=====
    //=====
    //=====
    // Redraw all elements
    //=====
    //=====

    if (pDoc->CElementList.IsEmpty() == false)
    {
        for (POSITION pos = pDoc->CElementList.GetHeadPosition();
pos != NULL; )
        {
            pDoc->CElementList.GetAt (pos) ->DrawOnDC (pDC);
            pDoc->CElementList.GetNext (pos);
        }
    }

    //=====
    //=====
    //=====
    // ***** APPLY GRAMMAR RULES *****
    //=====
    //=====

    //=====
    //=====
    // Check for uniqueness of GivenName of FUNCTIONS
    //=====
    //=====

    if (pDoc->CFunctionList.IsEmpty() == false)
    {
        for (POSITION pos = pDoc->CFunctionList.GetHeadPosition();
pos != pDoc->CFunctionList.GetTailPosition(); )
        {
            if (pDoc->CFunctionList.GetAt (pos) ->GivenName      ==
pDoc->CFunctionList.GetTail () ->GivenName)
            {

```

```

        AfxMessageBox("ILLEGAL NAMING :: ABORTING
INSTANCE.\n\n***** " +
                                pDoc-
>CFunctionList.GetAt(pos)->GivenName +
                                " *****\n\nFunction
names must be unique.");
        DeleteElement(pDoc->CFunctionList.GetTail());
        return;
    }
    pDoc->CFunctionList.GetNext(pos);
}

//=====
// Check for uniqueness of GivenName of ENV
//=====
if (pDoc->CEnvList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CEnvList.GetHeadPosition(); pos
!= pDoc->CEnvList.GetTailPosition(); )
    {
        if (pDoc->CEnvList.GetAt(pos)->GivenName == pDoc-
>CEnvList.GetTail()->GivenName)
        {
            AfxMessageBox("ILLEGAL NAMING :: ABORTING
INSTANCE.\n\n***** " +
                                pDoc-
>CEnvList.GetAt(pos)->GivenName +
                                " *****\n\nEnvironment
names must be unique.");
            DeleteElement(pDoc->CEnvList.GetTail());
            return;
        }
        pDoc->CEnvList.GetNext(pos);
    }
}

//=====
// Check for uniqueness of GivenName of MATERIAL
//=====
if (pDoc->CMaterialList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CMaterialList.GetHeadPosition();
pos != pDoc->CMaterialList.GetTailPosition(); )
    {
        if (pDoc->CMaterialList.GetAt(pos)->GivenName ==
pDoc->CMaterialList.GetTail()->GivenName)
        {

```

```

        AfxMessageBox("ILLEGAL NAMING :: ABORTING
INSTANCE.\n\n***** " +
                                pDoc-
>CMaterialList.GetAt(pos)->GivenName +
                                " *****\n\nMaterial
flow names must be unique.");
        DeleteElement(pDoc->CMaterialList.GetTail());
        return;
    }
    pDoc->CMaterialList.GetNext(pos);
}

//=====
// Check for uniqueness of GivenName of ENERGY
//=====
if (pDoc->CEnergyList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition();
pos != pDoc->CEnergyList.GetTailPosition(); )
    {
        if (pDoc->CEnergyList.GetAt(pos)->GivenName == pDoc-
>CEnergyList.GetTail()->GivenName)
        {
            AfxMessageBox("ILLEGAL NAMING :: ABORTING
INSTANCE.\n\n***** " +
                                pDoc-
>CEnergyList.GetAt(pos)->GivenName +
                                " *****\n\nEnergy flow
names must be unique.");
            DeleteElement(pDoc->CEnergyList.GetTail());
            return;
        }
        pDoc->CEnergyList.GetNext(pos);
    }
}

//=====
// Check for uniqueness of GivenName of NEW SIGNAL
//=====
if (pDoc->CSignalList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CSignalList.GetHeadPosition();
pos != pDoc->CSignalList.GetTailPosition(); )
    {
        if (pDoc->CSignalList.GetAt(pos)->GivenName == pDoc-
>CSignalList.GetTail()->GivenName)
        {

```

```

        AfxMessageBox("ILLEGAL NAMING :: ABORTING
INSTANCE.\n\n***** " +
        pDoc->CSignalList.GetAt(pos)->GivenName +
        " *****\n\nSignal flow
names must be unique.");
        DeleteElement(pDoc->CSignalList.GetTail());
        return;
    }
    pDoc->CSignalList.GetNext(pos);
}

//=====
=====
// Check for the Env-Flow-Env construct
// Check for head node = tail node construct
// Check for the Double-Carrier construct
// Check for Wrong Carrier Hierarchy, e.g., E carrying M
// Check for Carried head != Carrier head construct
//=====
=====

if (pDoc->CEdgeList.IsEmpty() == false)
{
    for (POSITION pos = pDoc->CEdgeList.GetHeadPosition();
((pos != NULL) && (GrammarCheckRequired)); )
    {
        //=====
        // Check for the Env-Flow-Env construct
        //=====
        if (ElementIsNode(pDoc->CEdgeList.GetAt(pos)-
>pTailElem) &&
            (pDoc->CEdgeList.GetAt(pos)->pTailElem == pDoc-
>CEdgeList.GetAt(pos)->pHeadElem))
        {
            GrammarCheckRequired = false; // This call is
very important.
            // Without it, the same instance is attempted
to delete multiple
            // times and the system crashes because it does
not find the
            // instance the second time around.
            AfxMessageBox(_T("ILLEGAL TOPOLOGY :: ABORTING
OPERATION. \n\nA flow cannot have the same head and tail node.\n\n(Get
real - This ain't no FunctionCAD)"));
            // There are two ways to create the Head = Tail
situation.
            // (1) At creation time - by selecting the same
node twice
            // (2) by dragging an existing signal end to
two Env's

```

```

// These three situations are addressed here.
    if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
        DeleteElement(pDoc->CEdgeList.GetTail());

    if (WhatToDo == ESCAPE) // Case 2
    {
        pDoc->CEdgeList.GetAt(pos)->pHeadElem =
pRememberHeadElement;
        pDoc->CEdgeList.GetAt(pos)->HeadPoint =
RememberHeadPoint;
        pDoc->CEdgeList.GetAt(pos)->pTailElem =
pRememberTailElement;
        pDoc->CEdgeList.GetAt(pos)->TailPoint =
RememberTailPoint;
    }

    if ((WhatToDo == ADD_ENV) || (WhatToDo ==
ADD_FUNCTION)) // Case 3
        DeleteElement(pDoc->CNodeList.GetTail());

    return;
}

//=====
// Check for the Env-Flow-Env construct
//=====
if (ElementIsEnv(pDoc->CEdgeList.GetAt(pos)-
>pTailElem) &&
    ElementIsEnv(pDoc->CEdgeList.GetAt(pos)-
>pHeadElem))
{
    GrammarCheckRequired = false;

    AfxMessageBox(_T("ILLEGAL TOPOLOGY :: ABORTING
OPERATION. \n\nA flow cannot connect to Env's.));

    // There are three ways to create the Env-flow-
Env construct:
    // (1) by adding a flow between two Env
instances,
    // (2) by dragging an existing flow end to two
Env's, and
    // (3) by adding an Env instance to the end of
a flow that
    // already has an Env at the other end. These
three situations
    // are addressed here.

    if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
        DeleteElement(pDoc->CEdgeList.GetTail());

```

```

        if (WhatToDo == ESCAPE)          // Case 2
        {
            pDoc->CEdgeList.GetAt(pos)->pHeadElem    =
pRememberHeadElement;
            pDoc->CEdgeList.GetAt(pos)->HeadPoint    =
RememberHeadPoint;
            pDoc->CEdgeList.GetAt(pos)->pTailElem    =
pRememberTailElement;
            pDoc->CEdgeList.GetAt(pos)->TailPoint    =
RememberTailPoint;
        }

        if ((WhatToDo == ADD_ENV) || (WhatToDo ==
ADD_FUNCTION)) // Case 3
            DeleteElement(pDoc->CNodeList.GetTail());

        return;
    }

    //=====
    // Check for the Double-Carrier construct
    //=====
    if (ElementIsEdge(pDoc->CEdgeList.GetAt(pos)-
>pTailElem) &&
        ElementIsEdge(pDoc->CEdgeList.GetAt(pos)-
>pHeadElem))
    {
        GrammarCheckRequired = false;
        AfxMessageBox(_T("ILLEGAL TOPOLOGY :: ABORTING
OPERATION. \n\nA flow cannot have two carriers."));

        // There are two ways of creating the double-
carrier construct:
        // (1) At the time of adding a flow, two other
flows can be selected
        // (2) by connecting the end of a flow to a
carrier, while the
        // other end already has a carrier
        // Both cases are addressed here.
        if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
            DeleteElement(pDoc->CEdgeList.GetTail());

        if (WhatToDo == ESCAPE)          // Case 2
        {
            pDoc->CEdgeList.GetAt(pos)->pHeadElem    =
pRememberHeadElement;
            pDoc->CEdgeList.GetAt(pos)->HeadPoint    =
RememberHeadPoint;
            pDoc->CEdgeList.GetAt(pos)->pTailElem    =
pRememberTailElement;

```

```

                                pDoc->CEdgeList.GetAt(pos)->TailPoint    =
RememberTailPoint;
                                }
                                if ((WhatToDo == ADD_ENV) || (WhatToDo ==
ADD_FUNCTION)) // Case 3
                                DeleteElement(pDoc->CNodeList.GetTail());

                                return;
                                }

                                //=====
                                // Check for Wrong Carrier Hierarchy
                                //=====
                                if (
                                    ((ElementIsMaterial(pDoc-
>CEdgeList.GetAt(pos))) &&
                                        ((ElementIsEdge(pDoc-
>CEdgeList.GetAt(pos)->pHeadElem) || (ElementIsEdge(pDoc-
>CEdgeList.GetAt(pos)->pTailElem))))
                                        ||
                                        ((ElementIsEnergy(pDoc->CEdgeList.GetAt(pos)))
&&
                                            ((ElementIsEnergy(pDoc-
>CEdgeList.GetAt(pos)->pTailElem) ||
                                                (ElementIsSignal(pDoc-
>CEdgeList.GetAt(pos)->pTailElem) ||
                                                    (ElementIsEnergy(pDoc-
>CEdgeList.GetAt(pos)->pHeadElem) ||
                                                        (ElementIsSignal(pDoc-
>CEdgeList.GetAt(pos)->pHeadElem))))
                                        ||
                                        ((ElementIsSignal(pDoc->CEdgeList.GetAt(pos)))
&&
                                            ((ElementIsEdge(pDoc-
>CEdgeList.GetAt(pos)->pHeadElem) ||
                                                ((ElementIsNode(pDoc-
>CEdgeList.GetAt(pos)->pTailElem) || (ElementIsSignal(pDoc-
>CEdgeList.GetAt(pos)->pTailElem))))
                                                )
                                            )
                                        {
                                            GrammarCheckRequired = false;

                                            CString* pCarrierMessage = new CString;

                                            if (ElementIsMaterial(pDoc-
>CEdgeList.GetAt(pos)))
                                                *pCarrierMessage = "Material cannot be
carried by another flow. No, not even by another Material.";
                                            if (ElementIsEnergy(pDoc-
>CEdgeList.GetAt(pos)))
                                                *pCarrierMessage = "Energy can be carried
by Material only. Not by another Energy, not by a Signal.";

```

```

        if (ElementIsSignal(pDoc->CEdgeList.GetAt(pos)))
            *pCarrierMessage = "Signal must be carried by a M or E. It can go ONLY from its carrier to ONLY a Node.";
            AfxMessageBox(_T("ILLEGAL CARRIER HIERACHY :: ABORTING OPERATION.\n\n") + *pCarrierMessage);
            delete pCarrierMessage;

            if ((WhatToDo == ADD_MATERIAL) || (WhatToDo == ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
                DeleteElement(pDoc->CEdgeList.GetTail());

            if (WhatToDo == ESCAPE) // Case 2
            {
                pDoc->CEdgeList.GetAt(pos)->pHeadElem = pRememberHeadElement;
                pDoc->CEdgeList.GetAt(pos)->HeadPoint = RememberHeadPoint;
                pDoc->CEdgeList.GetAt(pos)->pTailElem = pRememberTailElement;
                pDoc->CEdgeList.GetAt(pos)->TailPoint = RememberTailPoint;
            }

            if ((WhatToDo == ADD_ENV) || (WhatToDo == ADD_FUNCTION)) // Case 3
                DeleteElement(pDoc->CNodeList.GetTail());

            return;
        }

        //=====
        // Check for Carried head != Carrier head construct
        FOR FLOW-to-NODE BAGGAGE
        //=====
        if (
            (ElementIsEdge(pDoc->CEdgeList.GetAt(pos)->pTailElem) &&
             (!ElementIsSignal(pDoc->CEdgeList.GetAt(pos))) &&
             (ElementIsNode(pDoc->CEdgeList.GetAt(pos)->pHeadElem) &&
              (pDoc->CEdgeList.GetAt(pos)->pHeadElem != pDoc->CEdgeList.GetAt(pos)->pTailElem->pHeadElem)
             )
            )
        {
            GrammarCheckRequired = false;
            AfxMessageBox(_T("ILLEGAL TOPOLOGY :: ABORTING OPERATION. \n\nA carried Energy flow can be input to ONLY the function \nthat inputs its carrier.));

```



```

        if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
            DeleteElement(pDoc->CEdgeList.GetTail());

        if (WhatToDo == ESCAPE) // Case 2
        {
            pDoc->CEdgeList.GetAt(pos)->pHeadElem =
pRememberHeadElement;
            pDoc->CEdgeList.GetAt(pos)->HeadPoint =
RememberHeadPoint;
            pDoc->CEdgeList.GetAt(pos)->pTailElem =
pRememberTailElement;
            pDoc->CEdgeList.GetAt(pos)->TailPoint =
RememberTailPoint;
        }

        if ((WhatToDo == ADD_ENV) || (WhatToDo ==
ADD_FUNCTION)) // Case 3
            DeleteElement(pDoc->CNodeList.GetTail());

        return;
    }

    //=====
    // Check for Carried head != Carrier head construct
for NODE-to-FLOW BAGGAGE
    //=====
    if (
        (ElementIsNode(pDoc-
>CEdgeList.GetAt(pos)->pTailElem) &&
        (ElementIsEdge(pDoc-
>CEdgeList.GetAt(pos)->pHeadElem) &&
        (pDoc->CEdgeList.GetAt(pos)->pTailElem !=
pDoc->CEdgeList.GetAt(pos)->pHeadElem->pTailElem)
        )
    {
        GrammarCheckRequired = false;
        AfxMessageBox(_T("ILLEGAL TOPOLOGY :: ABORTING
OPERATION. \n\nA carried Energy flow can be added to a flow ONLY by the
function \nthat outputs its carrier."));

        if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
            DeleteElement(pDoc->CEdgeList.GetTail());

        if (WhatToDo == ESCAPE) // Case 2
        {
            pDoc->CEdgeList.GetAt(pos)->pHeadElem =
pRememberHeadElement;
            pDoc->CEdgeList.GetAt(pos)->HeadPoint =
RememberHeadPoint;
            pDoc->CEdgeList.GetAt(pos)->pTailElem =
pRememberTailElement;

```

```

pDoc->CEdgeList.GetAt(pos)->TailPoint =
RememberTailPoint;
    }
    if ((WhatToDo == ADD_ENV) || (WhatToDo ==
ADD_FUNCTION)) // Case 3
        DeleteElement(pDoc->CNodeList.GetTail());
    return;
}
pDoc->CEdgeList.GetNext(pos);
}
}

//=====
// Check for grammar rules of the CONVERT_E Template
//=====

if (pDoc->CConvert_E_Function_List.IsEmpty() == false)
{
    for (POSITION pos = pDoc-
>CConvert_E_Function_List.GetHeadPosition(); ((pos != NULL) &&
(GrammarCheckRequired)); )
    {
        bool* pThisConvEHasNoInputE = new bool;
        bool* pThisConvEHasNoOutEResidual = new bool;
        bool* pThisConvEHasMAttached = new bool;
        bool* pThisConvDoesNotConvertAnything = new bool;

        *pThisConvEHasNoInputE = true;
        *pThisConvEHasNoOutEResidual = true;
        *pThisConvEHasMAttached = false;
        *pThisConvDoesNotConvertAnything = true;

        for (POSITION pos1 = pDoc-
>CEnergyList.GetHeadPosition(); pos1 != NULL; )
        {
            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem ==
pDoc->CConvert_E_Function_List.GetAt(pos))
                *pThisConvEHasNoInputE = false;

            if ((pDoc->CEnergyList.GetAt(pos1)->pTailElem
== pDoc->CConvert_E_Function_List.GetAt(pos)) &&
(pDoc->CEnergyList.GetAt(pos1)-
>IsResidual))
                *pThisConvEHasNoOutEResidual = false;

            for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
                {

```

```

        if ((pDoc->CEnergyList.GetAt(pos1)-
>pHeadElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&
        (pDoc->CEnergyList.GetAt(pos2)-
>pTailElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&
        (pDoc->CEnergyList.GetAt(pos2)-
>IsResidual == false) &&
        (pDoc->CEnergyList.GetAt(pos1)-
>EnergyTypeName != pDoc->CEnergyList.GetAt(pos2)->EnergyTypeName))
            *pThisConvDoesNotConvertAnything =
false;

        pDoc->CEnergyList.GetNext(pos2);
    }

    pDoc->CEnergyList.GetNext(pos1);
}

for (POSITION pos1 = pDoc-
>CMaterialList.GetHeadPosition(); pos1 != NULL; )
{
    if ((pDoc->CMaterialList.GetAt(pos1)->pHeadElem
== pDoc->CConvert_E_Function_List.GetAt(pos)) ||
        (pDoc->CMaterialList.GetAt(pos1)-
>pTailElem == pDoc->CConvert_E_Function_List.GetAt(pos)))
        *pThisConvEHasMAttached = true;

    pDoc->CMaterialList.GetNext(pos1);
}

if ((*pThisConvEHasNoInputE) ||
(*pThisConvEHasNoOutEResidual) ||
(*pThisConvEHasMAttached) ||
(*pThisConvDoesNotConvertAnything))
{
    GrammarCheckRequired = false;

    CString* pLine1 = new CString;
    CString* pLine2 = new CString;
    CString* pLine3 = new CString;
    CString* pLine4 = new CString;
    CString* pLine5 = new CString;

    *pLine1 = "\n(1) Minimum one E in, ";
    *pLine2 = "\n(2) Minimum one E out, ";
    *pLine3 = "\n(3) Minimum one residual E out,
which could be the only output, and";
    *pLine4 = "\n(3) At least one pair of Input E
and Output E must be of different subtype.";
    *pLine5 = "\n\n(5) No Material flows are
allowed.";
}

```

```

        AfxMessageBox(_T("ILLEGAL TEMPLATE :: ABORTING
OPERATION. \n\nConvert_E needs:") + *pLine1 + *pLine2 + *pLine3 +
*pLine4 + *pLine5);

        delete pLine1;
        delete pLine2;
        delete pLine3;
        delete pLine4;
        delete pLine5;

        if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
            DeleteElement(pDoc->CEdgeList.GetTail());

        else if ((WhatToDo == ESCAPE) &&
(ElementIsEdge(pSelectedElement))) // Case 2
        {
            pSelectedElement->pHeadElem           =
pRememberHeadElement;
            pSelectedElement->HeadPoint           =
RememberHeadPoint;
            pSelectedElement->pTailElem           =
pRememberTailElement;
            pSelectedElement->TailPoint           =
RememberTailPoint;
        }

        else if (pSelectedElement == NULL) //      When
template violation occurs because of DELETING A REQUIRED FLOW
        {
            AfxMessageBox(_T("This action will delete
the template function, \nsince a necessary condition is violated.
\n\n(Sorry - Can't Undo.)"));
            DeleteElement(pDoc->CConvert_E_Function_List.GetAt(pos));
        }

        return;
    }

    delete pThisConvEHasNoInputE;
    delete pThisConvEHasNoOuterResidual;
    delete pThisConvEHasMAttached;
    delete pThisConvDoesNotConvertAnything;

    pDoc->CConvert_E_Function_List.GetNext(pos);
}

}

//=====
=====
// Check for grammar rules of the Conduct_E Template

```

```

//=====
=====

    if (pDoc->CConduct_E_Function_List.IsEmpty() == false)
    {
        for (POSITION pos = pDoc->CConduct_E_Function_List.GetHeadPosition(); ((pos != NULL) &&
(GrammarCheckRequired)); )
        {
            bool* pThisConductHasExactlyOneInputE = new bool;
            bool* pThisConductHasMultipleUsableOutputE = new
bool;
            bool* pThisConductCausesTypeChangeOfUsableOutput =
new bool;
            bool* pThisConductHasNoResidualOutputE = new bool;
            bool* pThisConductHasNoOutputOfSameTypeAsInput = new
bool;
            bool* pThisConductEHasMAttached = new bool;

            *pThisConductHasExactlyOneInputE = false;
            *pThisConductHasMultipleUsableOutputE = false;
            *pThisConductCausesTypeChangeOfUsableOutput = false;
            *pThisConductHasNoResidualOutputE = true;
            *pThisConductHasNoOutputOfSameTypeAsInput = true;
            *pThisConductEHasMAttached = false;

            int* pInputECount = new int;
            *pInputECount = 0;
            int* pUsableOutputECount = new int;
            *pUsableOutputECount = 0;

            for (POSITION pos1 = pDoc->CEnergyList.GetHeadPosition(); pos1 != NULL; )
            {
                if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem ==
pDoc->CConduct_E_Function_List.GetAt(pos))
                    *pInputECount = *pInputECount + 1;

                if ((pDoc->CEnergyList.GetAt(pos1)->pTailElem
== pDoc->CConduct_E_Function_List.GetAt(pos)) &&
(pDoc->CEnergyList.GetAt(pos1)-
>IsResidual == false))
                    *pUsableOutputECount
                    =
                    *pUsableOutputECount + 1;

                for (POSITION pos2 = pDoc->CEnergyList.GetHeadPosition(); pos2 != NULL; )
                {
                    if ((pDoc->CEnergyList.GetAt(pos1)-
>pHeadElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&
(pDoc->CEnergyList.GetAt(pos2)-
>pTailElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&

```

```

                                (pDoc->CEnergyList.GetAt(pos2)-
>IsResidual == false) &&
                                (pDoc->CEnergyList.GetAt(pos1)-
>EnergyTypeName != pDoc->CEnergyList.GetAt(pos2)->EnergyTypeName)

    *pThisConductCausesTypeChangeOfUsableOutput = true;

                                if      ((pDoc->CEnergyList.GetAt(pos1)-
>pHeadElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&
                                (pDoc->CEnergyList.GetAt(pos2)-
>pTailElem == pDoc->CConvert_E_Function_List.GetAt(pos)) &&
                                (pDoc->CEnergyList.GetAt(pos1)-
>EnergyTypeName == pDoc->CEnergyList.GetAt(pos2)->EnergyTypeName))

    *pThisConductHasNoOutputOfSameTypeAsInput = false;

                                pDoc->CEnergyList.GetNext(pos2);
                                }

                                if      ((pDoc->CEnergyList.GetAt(pos1)->pTailElem
== pDoc->CConduct_E_Function_List.GetAt(pos)) &&
                                (pDoc->CEnergyList.GetAt(pos1)-
>IsResidual == true))
                                *pThisConductHasNoResidualOutputE =
false;

                                pDoc->CEnergyList.GetNext(pos1);
                                }

    if (*pInputECount == 1)
        *pThisConductHasExactlyOneInputE = true;
    if (*pUsableOutputECount > 1)
        *pThisConductHasMultipleUsableOutputE = true;

    delete pInputECount;
    delete pUsableOutputECount;

    for      (POSITION      pos1      =      pDoc-
>CMaterialList.GetHeadPosition(); pos1 != NULL; )
    {
        if      ((pDoc->CMaterialList.GetAt(pos1)->pHeadElem
== pDoc->CConduct_E_Function_List.GetAt(pos)) ||
                (pDoc->CMaterialList.GetAt(pos1)-
>pTailElem == pDoc->CConduct_E_Function_List.GetAt(pos)))
            *pThisConductEHasMAttached = true;

        pDoc->CMaterialList.GetNext(pos1);
    }

    if      ((!(*pThisConductHasExactlyOneInputE)) ||
(*pThisConductHasMultipleUsableOutputE) ||

```

```

        (*pThisConductCausesTypeChangeOfUsableOutput)
|| (*pThisConductHasNoResidualOutputE) ||
        (*pThisConductHasNoOutputOfSameTypeAsInput) ||
(*pThisConductEHasMAttached)
    {
        GrammarCheckRequired = false;

        CString* pLine1 = new CString;
        CString* pLine2 = new CString;
        CString* pLine3 = new CString;
        CString* pLine4 = new CString;
        CString* pLine5 = new CString;
        CString* pLine6 = new CString;

        *pLine1 = "\n(1) Exactly one input E, ";
        *pLine2 = "\n(2) Maximum one usable output E
(if more than one, then it is branching), ";
        *pLine3 = "\n(3) The usable output E, if
present, must be of the same type as the input E,";
        *pLine4 = "\n(4) Minimum one residual output E,
";
        *pLine5 = "\n(5) At least one output E must be
of the same type as the input E, and";
        *pLine6 = "\n\n(6) No Material flows are
allowed.";

        AfxMessageBox(_T("ILLEGAL TEMPLATE :: ABORTING
OPERATION. \n\nConduct_E needs:") + *pLine1 + *pLine2 + *pLine3 +
*pLine4 + *pLine5 + *pLine6);

        delete pLine1;
        delete pLine2;
        delete pLine3;
        delete pLine4;
        delete pLine5;
        delete pLine6;

        if ((WhatToDo == ADD_MATERIAL) || (WhatToDo ==
ADD_ENERGY) || (WhatToDo == ADD_SIGNAL)) // Case 1
            DeleteElement(pDoc->CEdgeList.GetTail());

        else if ((WhatToDo == ESCAPE) &&
(ElementIsEdge(pSelectedElement))) // Case 2
        {
            pSelectedElement->pHeadElem =
pRememberHeadElement;
            pSelectedElement->HeadPoint =
RememberHeadPoint;
            pSelectedElement->pTailElem =
pRememberTailElement;
            pSelectedElement->TailPoint =
RememberTailPoint;
        }
    }

```

```

                else if (pSelectedElement == NULL) //      When
template violation occurs because of DELETING A REQUIRED FLOW
                {
                    AfxMessageBox(_T("This action will delete
the template function, \nsince a necessary condition is violated.
\n\n(Sorry - Can't Undo.)"));
                    DeleteElement(pDoc-
>CConduct_E_Function_List.GetAt(pos));
                }/**/

                return;
            }

            delete pThisConductHasExactlyOneInputE;
            delete pThisConductHasMultipleUsableOutputE;
            delete pThisConductCausesTypeChangeOfUsableOutput;
            delete pThisConductHasNoResidualOutputE;
            delete pThisConductHasNoOutputOfSameTypeAsInput;
            delete pThisConductEHasMAttached;

            pDoc->CConduct_E_Function_List.GetNext(pos);
        }
    }

    GrammarCheckRequired = true;
}

//
=====
//
=====
// PRINTING FUNCTIONS
//
=====
//
=====

BOOL CConModView::OnPreparePrinting(CPrintInfo* pInfo)
{
    // default preparation
    return DoPreparePrinting(pInfo);
}

void CConModView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

```



```

void CConModView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add cleanup after printing
}

//
=====
//
=====
// DIAGNOSTICS AND EXCEPTION HANDLING
//
=====
//
=====

#ifdef _DEBUG
void CConModView::AssertValid() const
{
    CView::AssertValid();
}

void CConModView::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

//
=====
//
=====
// DOCUMENT POINTER
//
=====
//
=====

CConModDoc* CConModView::GetDocument() const // non-debug version is
inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CConModDoc)));
    return (CConModDoc*)m_pDocument;
}
#endif // _DEBUG

```

```

//
=====
//
=====
// STANDARD TOOLBAR EVENT HANDLER - FILE SAVE
//
=====
//
=====

void CConModView::Handler_SaveFile(void)
{
    AfxMessageBox(_T("Save File."));
}

//
=====
//
=====
// MAIN MENU - REASONING OPTION MESSAGE HANDLER FUNCTIONS
//
=====
//
=====

void CConModView::OnQualitativeConservation()
{
    ReasoningOption = QUALITATIVE_CONSERVATION;
    AfxMessageBox(_T("Reasoning Switched to: QUALITATIVE
CONSERVATION."));
}

void CConModView::OnQualitativeIrreversibility()
{
    ReasoningOption = QUALITATIVE_IRREVERSIBILITY;
    AfxMessageBox(_T("Reasoning Switched to: QUALITATIVE
IRREVERSIBILITY."));
}

void CConModView::OnQuantitativeEfficiency()
{
    ReasoningOption = QUANTITATIVE_EFFICIENCY;
    AfxMessageBox(_T("Reasoning Switched to: EFFICIENCY."));
}

void CConModView::OnQuantitativePowerRequired()

```

```

{
    ReasoningOption = QUANTITATIVE_POWERREQUIRED;
    AfxMessageBox(_T("Reasoning Switched to: POWER REQUIRED.));
}

//
=====
//
=====
// PRIMITIVES TOOLBAR EVENT HANDLER FUNCTIONS - ONLY FOR SETTING "WHAT
TO DO"
//
=====
//
=====

void CConModView::Handler_AddFunction(void)
{
    WhatToDo = ADD_FUNCTION;
}

void CConModView::Handler_AddMaterial(void)
{
    WhatToDo = ADD_MATERIAL;
}

void CConModView::Handler_AddEnergy(void)
{
    WhatToDo = ADD_ENERGY;
}

void CConModView::Handler_AddSignal(void)
{
    WhatToDo = ADD_SIGNAL;
}

void CConModView::Handler_AddEnv(void)
{
    WhatToDo = ADD_ENV;
}

//
=====
//
=====
// FEATURES TOOLBAR EVENT HANDLER FUNCTIONS - ONLY FOR SETTING "WHAT TO
DO"

```

```

//
=====
//
=====

void CConModView::Handler_AddConvert_E_Template(void)
{
    WhatToDo = ADD_CONVERT_E_TEMPLATE;
}

void CConModView::Handler_AddConduct_E_Template(void)
{
    WhatToDo = ADD_CONDUCT_E_TEMPLATE;
}

void CConModView::Handler_AddEnergize_M_Template(void)
{
    WhatToDo = ADD_ENERGIZE_M_TEMPLATE;
}

void CConModView::Handler_AddDistribute_E_Template(void)
{
    WhatToDo = ADD_DISTRIBUTE_E_TEMPLATE;
}

void CConModView::Handler_AddDeEn_M_Template(void)
{
    WhatToDo = ADD_DEEN_M_TEMPLATE;
}

//
=====
//
=====
// REASONING TOOLBAR EVENT HANDLER FUNCTIONS - ONLY FOR SETTING "WHAT
TO DO"
//
=====
//
=====

void CConModView::Handler_Qualitative(void)
{
    ComposeQualitativeMessage();
}

void CConModView::Handler_Quantitative(void)

```

```

{
    ComposeQuantitativeMessage();
}

//
=====
//
=====
// EDIT TOOLBAR EVENT HANDLER FUNCTIONS
//
=====
//
=====

void CConModView::Handler_EditCut()
{
    if (pSelectedElement == NULL)
        return;
    if (WhatToDo == ESCAPE)
    {
        CConModDoc* pDoc = GetDocument();
        DetachEdgesFromElement(pSelectedElement);
        DeleteElement(pSelectedElement);
        pSelectedElement = NULL;           // Resets the pointer
to NULL
    }

    //OnDraw(this->GetDC());
};
//
=====
//
=====
// MOUSE EVENT HANDLER FUNCTIONS - CALLS THE APPROPRIATE INSTANCE-
ADDING FNC.
//
=====
//
=====

void CConModView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CConModDoc* pDoc = GetDocument();

    MouseLDownPoint = point;
}

```

```

        LButtonIsDown = TRUE;

        switch (WhatToDo)
        {
        case ESCAPE:
            if (ElementIsEdge(pSelectedElement))
            {
                // Remember head topology, in case you have to revert back
to this state.
                // This condition may arise if the new topology after move
/ connect is
                // unacceptable by the grammar rules, in which case the
OnDraw function
                // reverts the topology to the older "REMEMBERED" one.

                RememberHeadPoint = pSelectedElement->HeadPoint;
                pRememberHeadElement = pSelectedElement->pHeadElem;
                RememberTailPoint = pSelectedElement->TailPoint;
                pRememberTailElement = pSelectedElement->pTailElem;
            }
            break;

        case ADD_FUNCTION:
            AddFunction();
            break;

        case ADD_ENV:
            AddEnv();
            break;

        case ADD_CONVERT_E_TEMPLATE:
            AddConvert_E_Template();
            break;

        case ADD_CONDUCT_E_TEMPLATE:
            AddConduct_E_Template();
            break;

        case ADD_ENERGIZE_M_TEMPLATE:
            AddEnergize_M_Template();
            break;

        case ADD_DISTRIBUTE_E_TEMPLATE:
            AddDistribute_E_Template();
            break;

        case ADD_DEEN_M_TEMPLATE:
            AddDeEn_M_Template();
            break;
        }
    }

void CConModView::OnLButtonUp(UINT nFlags, CPoint point)

```

```

{
    MouseLUpPoint = point;
    CConModDoc* pDoc = GetDocument();

    switch (WhatToDo)
    {
    case ESCAPE:
        MoveConnect();
        break;

    case ADD_MATERIAL:
        AddMaterial();
        break;

    case ADD_ENERGY:
        AddEnergy();
        break;

    case ADD_SIGNAL:
        AddSignal();
        break;
    }

    LButtonIsDown = FALSE;
}

void CConModView::OnRButtonDown(UINT nFlags, CPoint point)
{
    MouseRDownPoint = point;
    RButtonIsDown = TRUE;

    CConModDoc* pDoc = GetDocument();

    // IF PreselectionList IS EMPTY, RIGHT CLICK WILL SET WhatToDo =
    ESCAPE
    // OTHERWISE, IF THE LIST IS FULL, IT SHOULD SCROLL THROUGH THAT
    LIST
    if (pDoc->PreselectionList.IsEmpty())
        WhatToDo = ESCAPE;
    else
        ScrollThroughPreselection();
}

void CConModView::OnRButtonUp(UINT nFlags, CPoint point)
{
    MouseRUpPoint = point;
    RButtonIsDown = FALSE;
}

void CConModView::OnMouseMove(UINT nFlags, CPoint point)
{
    MouseMovePoint = point;
}

```

```

// FOR ALL WHATTODO's, IF BOTH BUTTONS ARE UP, MOUSE MOVEMENT
WILL PRESELECT ELEMENTS

    if ((!LButtonIsDown) && (!RButtonIsDown))
        Preselect(&point);

    switch (WhatToDo)
    {
        case ESCAPE:
            if (LButtonIsDown && pSelectedElement != NULL)
                MoveConnectDynamic();
            break;

        case ADD_MATERIAL:
            if ((LButtonIsDown) && (!RButtonIsDown))
                AddEdge_Dynamic();
            break;

        case ADD_ENERGY:
            if ((LButtonIsDown) && (!RButtonIsDown))
                AddEdge_Dynamic();
            break;

        case ADD_SIGNAL:
            if ((LButtonIsDown) && (!RButtonIsDown))
                AddEdge_Dynamic();
            break;
    }
}

void CConModView::OnLButtonDbLClk(UINT nFlags, CPoint point)
{
    CConModDoc* pDoc = this->GetDocument();

    if (pSelectedElement == NULL)
        ComposeQualitativeMessage();
    else if (ElementIsFunction(pSelectedElement))
        pDoc->CFunctionList.GetAt(FunctionIndexInFunctionList)-
>DoModal();
    else if (ElementIsEnv(pSelectedElement))
        pDoc->CEnvList.GetAt(EnvIndexInEnvList)->DoModal();
    else if (ElementIsMaterial(pSelectedElement))
        pDoc->CMaterialList.GetAt(MaterialIndexInMaterialList)-
>DoModal();
    else if (ElementIsEnergy(pSelectedElement))
        pDoc->CEnergyList.GetAt(EnergyIndexInEnergyList)-
>DoModal();
    else if (ElementIsSignal(pSelectedElement))
        pDoc->CSignalList.GetAt(SignalIndexInSignalList)-
>DoModal();
}

void CConModView::OnMButtonUp(UINT nFlags, CPoint point)

```



```

{
    //if ((this->ElementIsEnergy(this->pSelectedElement)) ||
    //    (this->ElementIsMaterial(this->pSelectedElement)))
    //    this->pSelectedElement->IsResidual = !(this->
    >pSelectedElement->IsResidual);
}

BOOL CConModView::OnEraseBkgnd(CDC* pDC)
{
    return FALSE;
}

//
=====
=====
//
=====
=====
// FUNCTIONS THAT ADD, DELETE, AND MODIFY INSTANCES IN THE MODEL
//
=====
=====
//
=====
=====

void CConModView::AddFunction()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    //=====
=====
    // Auto-increment the name counter
    //=====
=====

    Counter_F = Counter_F + 1;
    CounterString.Format(_T("%d"), Counter_F);

    if ((ElementIsEdge(pSelectedElement)) && (pSelectedElement-
    >GrabHandle == TAIL)) // Grabbed an edge at tail
    {
        CFunction* NewCFunction = new CFunction(NULL,
        SnapToGrid(pSelectedElement->TailPoint), &CounterString);
        pSelectedElement->pTailElem = NewCFunction;
        pDoc->CElementList.AddTail(NewCFunction);
        pDoc->CNodeList.AddTail(NewCFunction);
        pDoc->CFunctionList.AddTail(NewCFunction);
    }

    if ((ElementIsEdge(pSelectedElement)) && (pSelectedElement-
    >GrabHandle == HEAD)) // Grabbed an edge at head
    {

```

```

        CFunction*      NewCFunction      =      new      CFunction(NULL,
SnapToGrid(pSelectedElement->HeadPoint), &CounterString);
        pSelectedElement->pHeadElem = NewCFunction;
        pDoc->CElementList.AddTail(NewCFunction);
        pDoc->CNodeList.AddTail(NewCFunction);
        pDoc->CFunctionList.AddTail(NewCFunction);
    }

    if (pSelectedElement == NULL)
    {
        CFunction*      NewCFunction      =      new      CFunction(NULL,
SnapToGrid(MouseLDownPoint), &CounterString);
        pDoc->CElementList.AddTail(NewCFunction);
        pDoc->CNodeList.AddTail(NewCFunction);
        pDoc->CFunctionList.AddTail(NewCFunction);
    }

    //OnDraw(this->GetDC());
    LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set                          // to TRUE, since a
click was made on the                // screen to add the
function. When the button            // is lifted, it is
usually in the Add Function          // Dialog, so the
graphics window does not know        // that L Button was
lifted. Therefore,                   // functions such as
Preselect misbehave.
}

void CConModView::AddEdge_Dynamic()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();
    CEdge* NewCEdge = new CEdge(MouseLDownPoint, MouseMovePoint);

    //NewCEdge->pTailElem = pTailElemDynamic;
    if (pTailElemDynamic != NULL)
    {
        NewCEdge->pTailElem = pTailElemDynamic;
        TailNodeSelected = true;
    }

    Preselect(&MouseMovePoint);
    NewCEdge->pHeadElem = pHeadElemDynamic;

    NewCEdge->DrawOnDC(this->GetDC());
    delete NewCEdge;
}

```

```

void CConModView::AddMaterial()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    Counter_M = Counter_M + 1;
    CounterString.Format(_T("%d"), Counter_M);

    CMaterial* NewCMaterial = new CMaterial(NULL, MouseLDownPoint,
    MouseLUpPoint, &CounterString, this->ReasoningOption);

    NewCMaterial->pTailElem = pTailElemDynamic;
    NewCMaterial->pHeadElem = pHeadElemDynamic;

    pDoc->CElementList.AddTail(NewCMaterial);
    pDoc->CEdgeList.AddTail(NewCMaterial);
    pDoc->CMaterialList.AddTail(NewCMaterial);
    //OnDraw(this->GetDC());

    // Clear up the temporary edge creation data for the next use
    pTailElemDynamic = NULL;
    pHeadElemDynamic = NULL;
    TailNodeSelected = false;
}

void CConModView::AddEnergy()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    Counter_E = Counter_E + 1;
    CounterString.Format(_T("%d"), Counter_E);

    CEnergy* NewCEnergy = new CEnergy(NULL, MouseLDownPoint,
    MouseLUpPoint, &CounterString, this->ReasoningOption);

    NewCEnergy->pTailElem = pTailElemDynamic;
    NewCEnergy->pHeadElem = pHeadElemDynamic;

    pDoc->CElementList.AddTail(NewCEnergy);
    pDoc->CEdgeList.AddTail(NewCEnergy);
    pDoc->CEnergyList.AddTail(NewCEnergy);
    //OnDraw(this->GetDC());

    // Clear up the temporary edge creation data for the next use
    pTailElemDynamic = NULL;
    pHeadElemDynamic = NULL;
    TailNodeSelected = false;
}

void CConModView::AddSignal()
{

```

```

    CConModDoc* pDoc = GetDocument();
    Invalidate();

    Counter_S = Counter_S + 1;
    CounterString.Format(_T("%d"), Counter_S);

    CSignal* NewCSignal = new CSignal(NULL, MouseLDownPoint,
    MouseLUpPoint, &CounterString);

    NewCSignal->pTailElem = pTailElemDynamic;
    NewCSignal->pHeadElem = pHeadElemDynamic;

    pDoc->CElementList.AddTail(NewCSignal);
    pDoc->CEdgeList.AddTail(NewCSignal);
    pDoc->CSignalList.AddTail(NewCSignal);
    //OnDraw(this->GetDC());

    // Clear up the temporary edge creation data for the next use
    pTailElemDynamic = NULL;
    pHeadElemDynamic = NULL;
    TailNodeSelected = false;
}

void CConModView::AddEnv()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    Counter_Env = Counter_Env + 1;
    CounterString.Format(_T("%d"), Counter_Env);

    if ((ElementIsEdge(pSelectedElement)) && (pSelectedElement-
    >GrabHandle == TAIL)) // Grabbed an edge at tail
    {
        CEnv* NewCEnv = new CEnv(NULL, SnapToGrid(pSelectedElement-
    >TailPoint), &CounterString);
        pSelectedElement->pTailElem = NewCEnv;
        pDoc->CElementList.AddTail(NewCEnv);
        pDoc->CNodeList.AddTail(NewCEnv);
        pDoc->CEnvList.AddTail(NewCEnv);
    }

    if ((ElementIsEdge(pSelectedElement)) && (pSelectedElement-
    >GrabHandle == HEAD)) // Grabbed an edge at head
    {
        CEnv* NewCEnv = new CEnv(NULL, SnapToGrid(pSelectedElement-
    >HeadPoint), &CounterString);
        pSelectedElement->pHeadElem = NewCEnv;
        pDoc->CElementList.AddTail(NewCEnv);
        pDoc->CNodeList.AddTail(NewCEnv);
        pDoc->CEnvList.AddTail(NewCEnv);
    }
}

```

```

        if (pSelectedElement == NULL)
        {
            CEnv* NewCEnv = new CEnv(NULL, SnapToGrid(MouseLDownPoint),
&CounterString);
            pDoc->CElementList.AddTail(NewCEnv);
            pDoc->CNodeList.AddTail(NewCEnv);
            pDoc->CEnvList.AddTail(NewCEnv);
        }

        //OnDraw(this->GetDC());
        LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set                               // to TRUE, since a
click was made on the                    // screen to add the
function. When the button                 // is lifted, it is
usually in the Add Function               // Dialog, so the
graphics window does not know            // that L Button was
lifted. Therefore,                        // functions such as
Preselect misbehave.
    }

void CConModView::AddConvert_E_Template()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    CString* pCounterString_F = new CString;
    CString* pCounterString_InE = new CString;
    CString* pCounterString_OutE = new CString;
    CString* pCounterString_OutE_Res = new CString;

    Counter_F ++;
    pCounterString_F->Format(_T("%d"), Counter_F);
    *pCounterString_F = *pCounterString_F + " [Conv_E]";
    Counter_E ++;
    pCounterString_InE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE_Res->Format(_T("%d"), Counter_E);

    if (pSelectedElement == NULL) // Create only in empty, white
space of the screen - otherwise more
        //attachment issues will arise
    {

```

```

        CConvert_E_Template*      NewCConvert_E_Template      =      new
CConvert_E_Template(NULL, SnapToGrid(MouseLDownPoint),
        pCounterString_F,                pCounterString_InE,
pCounterString_OutE, pCounterString_OutE_Res);

        pDoc->CTemplateList.AddTail(NewCConvert_E_Template);
        pDoc-
>CConvert_E_Template_List.AddTail(NewCConvert_E_Template);

        pDoc->CElementList.AddTail(NewCConvert_E_Template-
>pFunctionBlock);
        pDoc->CNodeList.AddTail(NewCConvert_E_Template-
>pFunctionBlock);
        pDoc->CFunctionList.AddTail(NewCConvert_E_Template-
>pFunctionBlock);
        pDoc-
>CConvert_E_Function_List.AddTail(NewCConvert_E_Template-
>pFunctionBlock); // Enables grammr checking

        pDoc->CElementList.AddTail(NewCConvert_E_Template-
>pEnergy_InE);
        pDoc->CEdgeList.AddTail(NewCConvert_E_Template-
>pEnergy_InE);
        pDoc->CEnergyList.AddTail(NewCConvert_E_Template-
>pEnergy_InE);

        pDoc->CElementList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE);
        pDoc->CEdgeList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE);
        pDoc->CEnergyList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE);

        pDoc->CElementList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE_Res);
        pDoc->CEdgeList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE_Res);
        pDoc->CEnergyList.AddTail(NewCConvert_E_Template-
>pEnergy_OutE_Res);
    }

    delete pCounterString_F;
    delete pCounterString_InE;
    delete pCounterString_OutE;
    delete pCounterString_OutE_Res;

    //OnDraw(this->GetDC());
    LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set
}

void CConModView::AddConduct_E_Template()

```

```

{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    CString* pCounterString_F = new CString;
    CString* pCounterString_InE = new CString;
    CString* pCounterString_OutE = new CString;
    CString* pCounterString_OutE_Res = new CString;

    Counter_F ++;
    pCounterString_F->Format(_T("%d"), Counter_F);
    *pCounterString_F = *pCounterString_F + " [Cond_E]";
    Counter_E ++;
    pCounterString_InE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE_Res->Format(_T("%d"), Counter_E);

    if (pSelectedElement == NULL) // Create only in empty, white
    space of the screen - otherwise more
        //attachment issues will arise
    {
        CConduct_E_Template*      NewCConduct_E_Template      = new
CConduct_E_Template(NULL, SnapToGrid(MouseLDownPoint),
        pCounterString_F,          pCounterString_InE,
pCounterString_OutE, pCounterString_OutE_Res);

        pDoc->CTemplateList.AddTail(NewCConduct_E_Template);
        pDoc->CConduct_E_Template_List.AddTail(NewCConduct_E_Template);

        pDoc->CElementList.AddTail(NewCConduct_E_Template->
pFunctionBlock);
        pDoc->CNodeList.AddTail(NewCConduct_E_Template->
pFunctionBlock);
        pDoc->CFunctionList.AddTail(NewCConduct_E_Template->
pFunctionBlock);
        pDoc->CConduct_E_Function_List.AddTail(NewCConduct_E_Template->
pFunctionBlock); // Enables grammr checking

        pDoc->CElementList.AddTail(NewCConduct_E_Template->
pEnergy_InE);
        pDoc->CEdgeList.AddTail(NewCConduct_E_Template->
pEnergy_InE);
        pDoc->CEnergyList.AddTail(NewCConduct_E_Template->
pEnergy_InE);

        pDoc->CElementList.AddTail(NewCConduct_E_Template->
pEnergy_OutE);
    }
}

```

```

        pDoc->CEdgeList.AddTail(NewCConduct_E_Template-
>pEnergy_OutE);
        pDoc->CEnergyList.AddTail(NewCConduct_E_Template-
>pEnergy_OutE);

        pDoc->CElementList.AddTail(NewCConduct_E_Template-
>pEnergy_OutE_Res);
        pDoc->CEdgeList.AddTail(NewCConduct_E_Template-
>pEnergy_OutE_Res);
        pDoc->CEnergyList.AddTail(NewCConduct_E_Template-
>pEnergy_OutE_Res);
    }

    delete pCounterString_F;
    delete pCounterString_InE;
    delete pCounterString_OutE;
    delete pCounterString_OutE_Res;

    //OnDraw(this->GetDC());
    LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set
}

void CConModView::AddEnergize_M_Template()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    CString* pCounterString_F = new CString;
    CString* pCounterString_InM = new CString;
    CString* pCounterString_OutM = new CString;
    CString* pCounterString_InE = new CString;
    CString* pCounterString_OutE = new CString;

    Counter_F ++;
    pCounterString_F->Format(_T("%d"), Counter_F);
    *pCounterString_F = *pCounterString_F + " [En_Mat]";
    Counter_M ++;
    pCounterString_InM->Format(_T("%d"), Counter_M);
    Counter_M ++;
    pCounterString_OutM->Format(_T("%d"), Counter_M);
    Counter_E ++;
    pCounterString_InE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE->Format(_T("%d"), Counter_E);

    if (pSelectedElement == NULL) // Create only in empty, white
space of the screen - otherwise more
        //attachment issues will arise
    {
        CEnergize_M_Template* NewCEnergize_M_Template = new
CEnergize_M_Template(NULL, SnapToGrid(MouseLDownPoint),

```



```

        pCounterString_F,                pCounterString_InM,
pCounterString_OutM, pCounterString_InE, pCounterString_OutE);

        pDoc->CTemplateList.AddTail (NewCEnergize_M_Template);
        pDoc-
>CEnergize_M_Template_List.AddTail (NewCEnergize_M_Template);

        pDoc->CElementList.AddTail (NewCEnergize_M_Template-
>pFunctionBlock);
        pDoc->CNodeList.AddTail (NewCEnergize_M_Template-
>pFunctionBlock);
        pDoc->CFunctionList.AddTail (NewCEnergize_M_Template-
>pFunctionBlock);
        pDoc-
>CEnergize_M_Function_List.AddTail (NewCEnergize_M_Template-
>pFunctionBlock); // Enables grammr checking

        pDoc->CElementList.AddTail (NewCEnergize_M_Template-
>pMaterial_InM);
        pDoc->CEdgeList.AddTail (NewCEnergize_M_Template-
>pMaterial_InM);
        pDoc->CMaterialList.AddTail (NewCEnergize_M_Template-
>pMaterial_InM);

        pDoc->CElementList.AddTail (NewCEnergize_M_Template-
>pMaterial_OutM);
        pDoc->CEdgeList.AddTail (NewCEnergize_M_Template-
>pMaterial_OutM);
        pDoc->CMaterialList.AddTail (NewCEnergize_M_Template-
>pMaterial_OutM);

        pDoc->CElementList.AddTail (NewCEnergize_M_Template-
>pEnergy_InE);
        pDoc->CEdgeList.AddTail (NewCEnergize_M_Template-
>pEnergy_InE);
        pDoc->CEnergyList.AddTail (NewCEnergize_M_Template-
>pEnergy_InE);

        pDoc->CElementList.AddTail (NewCEnergize_M_Template-
>pEnergy_OutE);
        pDoc->CEdgeList.AddTail (NewCEnergize_M_Template-
>pEnergy_OutE);
        pDoc->CEnergyList.AddTail (NewCEnergize_M_Template-
>pEnergy_OutE);

    }

    delete pCounterString_F;
    delete pCounterString_InM ;
    delete pCounterString_OutM;
    delete pCounterString_InE ;
    delete pCounterString_OutE;

```

```

        //OnDraw(this->GetDC());
        LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set
    }

void CConModView::AddDistribute_E_Template()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    CString* pCounterString_F = new CString;
    CString* pCounterString_InE = new CString;
    CString* pCounterString_OutE1 = new CString;
    CString* pCounterString_OutE2 = new CString;

    Counter_F ++;
    pCounterString_F->Format(_T("%d"), Counter_F);
    *pCounterString_F = *pCounterString_F + " [Dist_E]";
    Counter_E ++;
    pCounterString_InE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE1->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE2->Format(_T("%d"), Counter_E);

    if (pSelectedElement == NULL) // Create only in empty, white
space of the screen - otherwise more
        //attachment issues will arise
    {
        CDistribute_E_Template* NewCDistribute_E_Template = new
CDistribute_E_Template(NULL, SnapToGrid(MouseLDownPoint),
            pCounterString_F,                pCounterString_InE,
pCounterString_OutE1, pCounterString_OutE2);

        pDoc->CTemplateList.AddTail(NewCDistribute_E_Template);
        pDoc-
>CDistribute_E_Template_List.AddTail(NewCDistribute_E_Template);

        pDoc->CElementList.AddTail(NewCDistribute_E_Template-
>pFunctionBlock);
        pDoc->CNodeList.AddTail(NewCDistribute_E_Template-
>pFunctionBlock);
        pDoc->CFunctionList.AddTail(NewCDistribute_E_Template-
>pFunctionBlock);
        pDoc-
>CDistribute_E_Function_List.AddTail(NewCDistribute_E_Template-
>pFunctionBlock); // Enables grammr checking

        pDoc->CElementList.AddTail(NewCDistribute_E_Template-
>pEnergy_InE);
        pDoc->CEdgeList.AddTail(NewCDistribute_E_Template-
>pEnergy_InE);
    }
}

```

```

        pDoc->CEnergyList.AddTail(NewCDistribute_E_Template-
>pEnergy_InE);

        pDoc->CElementList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE1);
        pDoc->CEdgeList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE1);
        pDoc->CEnergyList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE1);

        pDoc->CElementList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE2);
        pDoc->CEdgeList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE2);
        pDoc->CEnergyList.AddTail(NewCDistribute_E_Template-
>pEnergy_OutE2);
    }

    delete pCounterString_F;
    delete pCounterString_InE ;
    delete pCounterString_OutE1;
    delete pCounterString_OutE2;

    //OnDraw(this->GetDC());
    LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set
}

void CConModView::AddDeEn_M_Template()
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    CString* pCounterString_F = new CString;
    CString* pCounterString_InM = new CString;
    CString* pCounterString_OutM = new CString;
    CString* pCounterString_InE = new CString;
    CString* pCounterString_OutE = new CString;

    Counter_F ++;
    pCounterString_F->Format(_T("%d"), Counter_F);
    *pCounterString_F = *pCounterString_F + " [DeEn_M]";
    Counter_M ++;
    pCounterString_InM->Format(_T("%d"), Counter_M);
    Counter_M ++;
    pCounterString_OutM->Format(_T("%d"), Counter_M);
    Counter_E ++;
    pCounterString_InE->Format(_T("%d"), Counter_E);
    Counter_E ++;
    pCounterString_OutE->Format(_T("%d"), Counter_E);
}

```

```

    if (pSelectedElement == NULL) // Create only in empty, white
    space of the screen - otherwise more
        //attachment issues will arise
    {
        CDeEn_M_Template*      NewCDeEn_M_Template      =      new
CDeEn_M_Template(NULL, SnapToGrid(MouseLDownPoint),
        pCounterString_F,          pCounterString_InM,
pCounterString_OutM, pCounterString_InE, pCounterString_OutE);

        pDoc->CTemplateList.AddTail(NewCDeEn_M_Template);
        pDoc->CDeEn_M_Template_List.AddTail(NewCDeEn_M_Template);

        pDoc->CElementList.AddTail(NewCDeEn_M_Template-
>pFunctionBlock);
        pDoc->CNodeList.AddTail(NewCDeEn_M_Template-
>pFunctionBlock);
        pDoc->CFunctionList.AddTail(NewCDeEn_M_Template-
>pFunctionBlock);
        pDoc-
>CEnergize_M_Function_List.AddTail(NewCDeEn_M_Template-
>pFunctionBlock); // Enables grammr checking

        pDoc->CElementList.AddTail(NewCDeEn_M_Template-
>pMaterial_InM);
        pDoc->CEdgeList.AddTail(NewCDeEn_M_Template-
>pMaterial_InM);
        pDoc->CMaterialList.AddTail(NewCDeEn_M_Template-
>pMaterial_InM);

        pDoc->CElementList.AddTail(NewCDeEn_M_Template-
>pMaterial_OutM);
        pDoc->CEdgeList.AddTail(NewCDeEn_M_Template-
>pMaterial_OutM);
        pDoc->CMaterialList.AddTail(NewCDeEn_M_Template-
>pMaterial_OutM);

        pDoc->CElementList.AddTail(NewCDeEn_M_Template-
>pEnergy_InE);
        pDoc->CEdgeList.AddTail(NewCDeEn_M_Template->pEnergy_InE);
        pDoc->CEnergyList.AddTail(NewCDeEn_M_Template-
>pEnergy_InE);

        pDoc->CElementList.AddTail(NewCDeEn_M_Template-
>pEnergy_OutE);
        pDoc->CEdgeList.AddTail(NewCDeEn_M_Template->pEnergy_OutE);
        pDoc->CEnergyList.AddTail(NewCDeEn_M_Template-
>pEnergy_OutE);
    }

    delete pCounterString_F;
    delete pCounterString_InM ;
    delete pCounterString_OutM;

```

```

delete pCounterString_InE ;
delete pCounterString_OutE;

//OnDraw(this->GetDC());
LButtonIsDown = FALSE;          // Without this line, LButtonIsDown
remains set
}

//
=====
//
=====
// FUNCTIONS TO SELECT AND UNSELECT OBJECTS FROM THE MODEL
//
=====
//
=====

void CConModView::Preselect(CPoint* pMouseTip)
{
    CConModDoc* pDoc = GetDocument();
    Invalidate();

    //
=====
//
// RESET THE EXISTING CONTAINERS OF PRESELECTION DATA AT EVERY
NEW CALL
//
=====
//
pDoc->PreselectionList.RemoveAll();
ScrollPosition = NULL;
pSelectedElement = NULL;

//
=====
//
// SPECIAL REQUIREMENT FOR EDGES - CLEAR OFF THE TEMPORARY HEAD
AND TAIL NODES
//
=====
//
if (!TailNodeSelected)
    pTailElemDynamic = NULL;
pHeadElemDynamic = NULL;

```

```

//
=====
// LOOK FOR PROXIMITY BETWEEN MOUSE TIP AND ALL CELEMENT
INSTANCES.
// FOR ALL PROXIMAL CELEMENT INSTANCES, HIGHLIGHT, ASSIGN
GRABHANDLE, AND ADD TO
// PRESELECTIONLIST (IF NOT ALREADY THERE).
// THREE TESTS FOR GRABHANDLE ARE NECESSARY TO PERFORM
THIS ACTION.
// IF NOT PROXIMAL, THEN UNHIGHLIHT, RESET GRABHANDLE, AND
REMOVE FROM
// PRESELECTIONLIST (IF NOT ALREADY REMOVED).
//
=====
=====

    if (!pDoc->CElementList.IsEmpty())
    {
        for (POSITION pos = pDoc->CElementList.GetHeadPosition();
pos != NULL; )
        {
            if (distance(*pMouseTip, pDoc-
>CElementList.GetAt(pos)->GeometricCenter) <= SELECTION_RADIUS)
            {
                pDoc->CElementList.GetAt(pos)->GrabHandle =
CENTER; // Applies to both nodes and edges
                Highlight(pDoc->CElementList.GetAt(pos));
                if (!pDoc->PreselectionList.Find(pDoc-
>CElementList.GetAt(pos)))
                    pDoc->PreselectionList.AddTail(pDoc-
>CElementList.GetAt(pos));
            }
            else if (distance(*pMouseTip, pDoc-
>CElementList.GetAt(pos)->HeadPoint) <= SELECTION_RADIUS)
            {
                pDoc->CElementList.GetAt(pos)->GrabHandle =
HEAD; // Applies to edges
                Highlight(pDoc->CElementList.GetAt(pos));
                if (!pDoc->PreselectionList.Find(pDoc-
>CElementList.GetAt(pos)))
                    pDoc->PreselectionList.AddTail(pDoc-
>CElementList.GetAt(pos));
            }
            else if (distance(*pMouseTip, pDoc-
>CElementList.GetAt(pos)->TailPoint) <= SELECTION_RADIUS)
            {
                pDoc->CElementList.GetAt(pos)->GrabHandle =
TAIL; // Applies to edges
                Highlight(pDoc->CElementList.GetAt(pos));
                if (!pDoc->PreselectionList.Find(pDoc-
>CElementList.GetAt(pos)))

```

```

                                pDoc->PreselectionList.AddTail (pDoc-
>CElementList.GetAt (pos));
                                }
                                else
                                {
                                    UnHighlight (pDoc->CElementList.GetAt (pos));
                                    pDoc->CElementList.GetAt (pos)->GrabHandle      =
NULL;
                                    if          (pDoc->PreselectionList.Find (pDoc-
>CElementList.GetAt (pos)))
                                        pDoc->PreselectionList.RemoveAt (pDoc-
>PreselectionList.Find (pDoc->CElementList.GetAt (pos)));          // (pDoc-
>CElementList.GetAt (pos));
                                    }

                                    pDoc->CElementList.GetNext (pos);
                                }
                                }

                                //
=====
// GET READY FOR SCROLLING:
// IF PreselectionList HAS THINGS IN IT, SELECT THE FIRST ITEM
AND SET
// ScrollPosition AS THE HEAD POSITION WITHIN THAT LIST.
// OTHERWISE, THE EXISTING NULL VALUES SET AT THE BEGINNING OF
THIS FUNCTION
// CALL WILL PREVAIL.
//
=====
if (!pDoc->PreselectionList.IsEmpty())
{
    SelectElement (pDoc->PreselectionList.GetHead());          //
stroes pSelectedElement
    ScrollPosition = pDoc->PreselectionList.GetHeadPosition();

    // SCPECIAL CASE - IF ADDING AN EDGE, STORE ITS temporary
TAIL and HEAD
    if    ((WhatToDo == ADD_ENERGY)    ||    (WhatToDo ==
ADD_MATERIAL) || (WhatToDo == ADD_SIGNAL))
    {
        if (LButtonIsDown)
            pHeadElemDynamic = pSelectedElement;
        else
            pTailElemDynamic = pSelectedElement;

        if (pHeadElemDynamic == pTailElemDynamic)
            pHeadElemDynamic = NULL;          // Prevents self-
cycling edges
    }
}

```

```

    }

    // Finally, redraw the screen
    //OnDraw(this->GetDC());
}

void CConModView::Highlight(CElement* pElement)
{
    pElement->IsHighlighted = true;
    pElement->IsSelected = false;
}

void CConModView::UnHighlight(CElement* pElement)
{
    pElement->IsHighlighted = false;
    pElement->IsSelected = false;
}

void CConModView::SelectElement(CElement* pElement)
{
    pSelectedElement = pElement;
    pElement->IsSelected = true;
    pElement->IsHighlighted = false;
}

void CConModView::ScrollThroughPreselection()
{
    CConModDoc* pDoc = GetDocument();

    // Reset the current selection to PRESELECTION_PEN_colors
    Highlight(pDoc->PreselectionList.GetAt(ScrollPosition));

    // If the tail of PreselectionList has arrived, start over at the
head
    if (ScrollPosition == pDoc->PreselectionList.GetTailPosition())
        ScrollPosition = pDoc->PreselectionList.GetHeadPosition();
    else
        pDoc->PreselectionList.GetNext(ScrollPosition);

    // Select the element at this incremented ScrollPosition
    SelectElement(pDoc->PreselectionList.GetAt(ScrollPosition));

    //OnDraw(this->GetDC());

    //
=====
=====
    // SCPECIAL CASE - IF ADDING EDGE, STORE ITS TAIL NODE and HEAD
NODE.
    // AN IDENTICAL IF STATEMENT IS ALSO USED IN Preselect, TO ENABLE
THE
    // SAME FEATURES IF TEH USER SELECTED THE FIRST SELECTED ELEMENT
WITHOUT

```



```

// SCROLLING.
//
=====
if (WhatToDo == ADD_ENERGY || WhatToDo == ADD_MATERIAL ||
WhatToDo == ADD_SIGNAL)
{
    if (LButtonIsDown)
        pHeadElemDynamic = pSelectedElement;
    else
        pTailElemDynamic = pSelectedElement;

    if (pHeadElemDynamic == pTailElemDynamic)
        pHeadElemDynamic = NULL; // Prevents self-
cycling edges
}
}

bool CConModView::ElementIsNode(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CNodeList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CNodeList.GetAt(pos) == pElement)
        {
            NodeIndexInNodeList = pos;
            return true;
        }
        pDoc->CNodeList.GetNext(pos);
    }
    return false;
}

bool CConModView::ElementIsFunction(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CFunctionList.GetAt(pos) == pElement)
        {
            FunctionIndexInFunctionList = pos;
            return true;
        }

        pDoc->CFunctionList.GetNext(pos);
    }

    return false;
}

```

```

bool CConModView::ElementIsEnv(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEnvList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CEnvList.GetAt(pos) == pElement)
        {
            EnvIndexInEnvList = pos;
            return true;
        }

        pDoc->CEnvList.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsEdge(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEdgeList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CEdgeList.GetAt(pos) == pElement)
        {
            EdgeIndexInEdgeList = pos;
            return true;
        }

        pDoc->CEdgeList.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsMaterial(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CMaterialList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CMaterialList.GetAt(pos) == pElement)
        {
            MaterialIndexInMaterialList = pos;
            return true;
        }

        pDoc->CMaterialList.GetNext(pos);
    }
}

```

```

    }

    return false;
}

bool CConModView::ElementIsEnergy(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CEnergyList.GetAt(pos) == pElement)
        {
            EnergyIndexInEnergyList = pos;
            return true;
        }

        pDoc->CEnergyList.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsSignal(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CSignalList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CSignalList.GetAt(pos) == pElement)
        {
            SignalIndexInSignalList = pos;
            return true;
        }

        pDoc->CSignalList.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsConvert_E_Function(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CConvert_E_Function_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CConvert_E_Function_List.GetAt(pos) == pElement)
        {

```

```

        Convert_E_Function_IndexInConvert_E_Function_List    =
pos;
        return true;
    }

    pDoc->CConvert_E_Function_List.GetNext(pos);
}

return false;
}

bool CConModView::ElementIsConvert_E_Template(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CConvert_E_Template_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CConvert_E_Template_List.GetAt(pos) == pElement)
        {
            Convert_E_Template_IndexInConvert_E_Template_List    =
pos;
            return true;
        }

        pDoc->CConvert_E_Template_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsConduct_E_Function(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CConduct_E_Function_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CConduct_E_Function_List.GetAt(pos) == pElement)
        {
            Conduct_E_Function_IndexInConduct_E_Function_List    =
pos;
            return true;
        }

        pDoc->CConduct_E_Function_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsConduct_E_Template(CElement* pElement)
{

```

```

        CConModDoc* pDoc = GetDocument();

        for (POSITION pos = pDoc->CConduct_E_Template_List.GetHeadPosition(); pos != NULL; )
        {
            if (pDoc->CConduct_E_Template_List.GetAt(pos) == pElement)
            {
                Conduct_E_Template_IndexInConduct_E_Template_List =
pos;
                return true;
            }

            pDoc->CConduct_E_Template_List.GetNext(pos);
        }

        return false;
    }

bool CConModView::ElementIsEnergize_M_Function(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEnergize_M_Function_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CEnergize_M_Function_List.GetAt(pos) == pElement)
        {
            Energize_M_Function_IndexInEnergize_M_Function_List =
pos;
            return true;
        }

        pDoc->CEnergize_M_Function_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsEnergize_M_Template(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEnergize_M_Template_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CEnergize_M_Template_List.GetAt(pos) == pElement)
        {
            Energize_M_Template_IndexInEnergize_M_Template_List =
pos;
            return true;
        }

        pDoc->CEnergize_M_Template_List.GetNext(pos);
    }
}

```

```

    }

    return false;
}

bool CConModView::ElementIsDistribute_E_Function(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CDistribute_E_Function_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CDistribute_E_Function_List.GetAt(pos) == pElement)
        {
            Distribute_E_Function_IndexInDistribute_E_Function_List = pos;
            return true;
        }

        pDoc->CDistribute_E_Function_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsDistribute_E_Template(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CDistribute_E_Template_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CDistribute_E_Template_List.GetAt(pos) == pElement)
        {
            Distribute_E_Template_IndexInDistribute_E_Template_List = pos;
            return true;
        }

        pDoc->CDistribute_E_Template_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsDeEn_M_Function(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CDeEn_M_Function_List.GetHeadPosition(); pos != NULL; )

```

```

    {
        if (pDoc->CDeEn_M_Function_List.GetAt(pos) == pElement)
        {
            DeEn_M_Function_IndexInDeEn_M_Function_List = pos;
            return true;
        }

        pDoc->CDeEn_M_Function_List.GetNext(pos);
    }

    return false;
}

bool CConModView::ElementIsDeEn_M_Template(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CDeEn_M_Template_List.GetHeadPosition(); pos != NULL; )
    {
        if (pDoc->CDeEn_M_Template_List.GetAt(pos) == pElement)
        {
            DeEn_M_Template_IndexInDeEn_M_Template_List = pos;
            return true;
        }

        pDoc->CDeEn_M_Template_List.GetNext(pos);
    }

    return false;
}

//
=====
//
=====
// FUNCTIONS TO EDIT OBJECTS WITHIN THE MODEL
//
=====
//
=====

void CConModView::MoveConnectDynamic() // Called by OnMouseMove
{
    CConModDoc* pDoc = GetDocument();
    GrammarCheckRequired = false; // This call is very important
- without it, // the grammar checks for topological error will
take effect DURING

```

```

// the move / connect operation BEFORE LIFTING
UP THE MOUSE L
// BUTTON and throw errors for topology that
the user has not
// committed to (by lifting mouse L button)

//
=====
// THIS IS A BASIC CHECK THAT AN ELEMENT IS SELECTED FOR MOVE OR
CONNECT.
// PRACTICALLY, THIS CHECK IS REDUNDANT, SINCE THE ONLY CALLING
FUNCTION
// OF THIS FUNCTION, OnMouseMove, MAKES SURE THAT AN ELEMENT IS
INDEED SELECTED.
//
=====
if (!LButtonIsDown || pSelectedElement == NULL)
    return;

//
=====
// IF AN EDGE IS ANCHORED ON ANY ONE SIDE, PREVENT MOVING IT BY
ITS CENTER
//
=====
if ((ElementIsEdge(pSelectedElement))
    &&
    (pSelectedElement->pHeadElem != NULL)
    ||
    (pSelectedElement->pTailElem != NULL))
    &&
    (pSelectedElement->GrabHandle == CENTER))
    return;

Invalidate();

//
=====
// MOVE NODES AND DOUBLY-DANLGING EDGES BY THE CENTER GRABHANDLE
//
=====
if (pSelectedElement->GrabHandle == CENTER) // Works
for nodes and edges with both ends dangling
{
    // First, compute the orientation and length of the arrow
using its existing

```



```

// center, tail and head points. This check will workk
even for the nodes, although
// that would not mean anything real. So, it is
unnecessary to check that the element
// is an edge.
    long    HalfDeltaX    =    pSelectedElement->HeadPoint.x    -
pSelectedElement->GeometricCenter.x;
    long    HalfDeltaY    =    pSelectedElement->HeadPoint.y    -
pSelectedElement->GeometricCenter.y;

// Then, move the center point. This moves nodes directly.
For edges, the ends
// need to be recalculated, as done next.
pSelectedElement->GeometricCenter = MouseMovePoint;

// Then re-compute the new head and tail poitns based on
the new center point.
    pSelectedElement->HeadPoint.x    =    pSelectedElement-
>GeometricCenter.x + HalfDeltaX;
    pSelectedElement->HeadPoint.y    =    pSelectedElement-
>GeometricCenter.y + HalfDeltaY;
    pSelectedElement->TailPoint.x    =    pSelectedElement-
>GeometricCenter.x - HalfDeltaX;
    pSelectedElement->TailPoint.y    =    pSelectedElement-
>GeometricCenter.y - HalfDeltaY;
}

//
=====
=====
// MOVE AND/OR CONNECT THE head POINT OF AN EDGE
//
=====
=====

    if    (ElementIsEdge (pSelectedElement)    &&    (pSelectedElement-
>GrabHandle == HEAD))
    {
        pSelectedElement->pHeadElem = NULL;

        for    (POSITION    pos = pDoc->CElementList.GetHeadPosition();
pos != NULL; )
        {
            if    (distance (MouseMovePoint,    pDoc-
>CElementList.GetAt (pos)->GeometricCenter) <= SELECTION_RADIUS)
            {
                Highlight (pDoc->CElementList.GetAt (pos));
                pSelectedElement->pHeadElem    =    pDoc-
>CElementList.GetAt (pos);
            }
            else    if    ((distance (MouseMovePoint,    pDoc-
>CElementList.GetAt (pos)->TailPoint) <= SELECTION_RADIUS) &&

```

```

        (ElementIsEdge (pDoc->CElementList.GetAt (pos)))
&&
        (pDoc->CElementList.GetAt (pos)->pHeadElem      !=
NULL) &&
        (pDoc->CElementList.GetAt (pos)->pTailElem      ==
NULL) )
    {
        Highlight (pDoc->CElementList.GetAt (pos));
        pSelectedElement->pHeadElem      =      pDoc-
>CElementList.GetAt (pos)->pHeadElem;
        pElementToBeDeleted              =      pDoc-
>CElementList.GetAt (pos);
    }
    else
    {
        UnHighlight (pDoc->CElementList.GetAt (pos));
        pSelectedElement->HeadPoint = MouseMovePoint;
        if      (pDoc->CElementList.GetAt (pos)      ==
pElementToBeDeleted)
            pElementToBeDeleted = NULL;
    }

    pDoc->CElementList.GetNext (pos);
}

//
=====
// MOVE AND/OR CONNECT THE tail POINT OF AN EDGE
//
=====

    if      (ElementIsEdge (pSelectedElement)      &&      (pSelectedElement-
>GrabHandle == TAIL))
    {
        pSelectedElement->pTailElem = NULL;

        for (POSITION pos = pDoc->CElementList.GetHeadPosition();
pos != NULL; )
        {
            if      (distance (MouseMovePoint,      pDoc-
>CElementList.GetAt (pos)->GeometricCenter) <= SELECTION_RADIUS)
            {
                Highlight (pDoc->CElementList.GetAt (pos));
                pSelectedElement->pTailElem      =      pDoc-
>CElementList.GetAt (pos);
            }
            else if      ((distance (MouseMovePoint,      pDoc-
>CElementList.GetAt (pos)->HeadPoint) <= SELECTION_RADIUS) &&
(ElementIsEdge (pDoc->CElementList.GetAt (pos)))
&&

```

```

        (pDoc->CElementList.GetAt(pos)->pTailElem    !=
NULL) &&
        (pDoc->CElementList.GetAt(pos)->pHeadElem    ==
NULL))
    {
        Highlight(pDoc->CElementList.GetAt(pos));
        pSelectedElement->pTailElem    =    pDoc->
>CElementList.GetAt(pos)->pTailElem;
        pElementToBeDeleted    =    pDoc->
>CElementList.GetAt(pos);
    }
    else
    {
        UnHighlight(pDoc->CElementList.GetAt(pos));
        pSelectedElement->TailPoint = MouseMovePoint;
        if    (pDoc->CElementList.GetAt(pos)    ==
pElementToBeDeleted)
            pElementToBeDeleted = NULL;
    }

    pDoc->CElementList.GetNext(pos);
}

// OnDraw(this->GetDC());    // Do NOT call OnDraw here - it
will fire the
// grammar checks before the move/connect is complete
}

void CConModView::MoveConnect()    // Called by OnLButtonUp,
when moving edges (ESCAPE)
{
    if (pSelectedElement == NULL)
        return;

    CConModDoc* pDoc = GetDocument();
    Invalidate();

    // SNAP THE NODES TO THE GRID AFTER MOVE IS OVER, WHEN L-BUTTON
IS LIFTED
    if (ElementIsNode(pSelectedElement))
        pSelectedElement->GeometricCenter    =
SnapToGrid(pSelectedElement->GeometricCenter);

    if (pElementToBeDeleted != NULL)
    {
        DeleteElement(pElementToBeDeleted);
        pElementToBeDeleted = NULL;
    }

    GrammarCheckRequired = true;

    //OnDraw(this->GetDC());

```

```

}

void CConModView::DetachEdgesFromElement(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();

    for (POSITION pos = pDoc->CEdgeList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pElement == pDoc->CEdgeList.GetAt(pos)->pHeadElem)
            pDoc->CEdgeList.GetAt(pos)->pHeadElem = NULL;
        if (pElement == pDoc->CEdgeList.GetAt(pos)->pTailElem)
            pDoc->CEdgeList.GetAt(pos)->pTailElem = NULL;

        pDoc->CEdgeList.GetNext(pos);
    }
}

void CConModView::DeleteElement(CElement* pElement)
{
    CConModDoc* pDoc = GetDocument();
    DetachEdgesFromElement(pElement);

    delete pElement; // Deletes the actual instance of the element
                    // pointed by pElement

    POSITION pos = pDoc->CElementList.Find(pElement);
    pDoc->CElementList.RemoveAt(pos); // Removes the pointer entry
from CElementList

    if (ElementIsNode(pElement))
        pDoc->CNodeList.RemoveAt(NodeIndexInNodeList); // Removes
the pointer entry from CNodeList

    if (ElementIsFunction(pElement))
        pDoc->CFunctionList.RemoveAt(FunctionIndexInFunctionList);
// Removes the pointer entry from CFunctionList

    if (ElementIsEnv(pElement))
        pDoc->CEnvList.RemoveAt(EnvIndexInEnvList); // Removes
the pointer entry from CFunctionList

    if (ElementIsEdge(pElement))
        pDoc->CEdgeList.RemoveAt(EdgeIndexInEdgeList); // Removes
the pointer entry from CEdgeList

    if (ElementIsMaterial(pElement))
        pDoc->CMaterialList.RemoveAt(MaterialIndexInMaterialList);
// Removes the pointer entry from CEdgeList

    if (ElementIsEnergy(pElement))
        pDoc->CEnergyList.RemoveAt(EnergyIndexInEnergyList); //
Removes the pointer entry from CEdgeList

```

```

        if (ElementIsSignal(pElement))
            pDoc->CSignalList.RemoveAt(SignalIndexInSignalList); //
Removes the pointer entry from CEdgeList

        if (ElementIsConvert_E_Template(pElement))
            pDoc-
>CConvert_E_Template_List.RemoveAt(Convert_E_Template_IndexInConvert_E_
Template_List); // Removes the pointer entry from CEdgeList

        if (ElementIsConvert_E_Function(pElement))
            pDoc-
>CConvert_E_Function_List.RemoveAt(Convert_E_Function_IndexInConvert_E_
Function_List); // Removes the pointer entry from CEdgeList

        if (ElementIsConduct_E_Template(pElement))
            pDoc-
>CConduct_E_Template_List.RemoveAt(Conduct_E_Template_IndexInConduct_E_
Template_List); // Removes the pointer entry from CEdgeList

        if (ElementIsConduct_E_Function(pElement))
            pDoc-
>CConduct_E_Function_List.RemoveAt(Conduct_E_Function_IndexInConduct_E_
Function_List); // Removes the pointer entry from CEdgeList

        if (ElementIsEnergize_M_Template(pElement))
            pDoc-
>CEnergize_M_Template_List.RemoveAt(Energize_M_Template_IndexInEnergize
_M_Template_List); // Removes the pointer entry from CEdgeList

        if (ElementIsEnergize_M_Function(pElement))
            pDoc-
>CEnergize_M_Function_List.RemoveAt(Energize_M_Function_IndexInEnergize
_M_Function_List); // Removes the pointer entry from CEdgeList

        if (ElementIsDistribute_E_Function(pElement))
            pDoc-
>CDistribute_E_Function_List.RemoveAt(Distribute_E_Function_IndexInDist
ribute_E_Function_List); // Removes the pointer entry from
CEdgeList

        if (ElementIsDistribute_E_Template(pElement))
            pDoc-
>CDistribute_E_Template_List.RemoveAt(Distribute_E_Template_IndexInDist
ribute_E_Template_List); // Removes the pointer entry from
CEdgeList

        if (ElementIsDeEn_M_Function(pElement))
            pDoc-
>CDeEn_M_Function_List.RemoveAt(DeEn_M_Function_IndexInDeEn_M_Function_
List); // Removes the pointer entry from CEdgeList

        if (ElementIsDeEn_M_Template(pElement))

```

```

        pDoc->CDeEn_M_Template_List.RemoveAt(DeEn_M_Template_IndexInDeEn_M_Template_List); // Removes the pointer entry from CEdgeList

        //for (POSITION pos = pDoc->CMaterialList.GetHeadPosition(); pos != NULL; )
        //{
            // for (POSITION pos1 = pDoc->CMaterialList.GetAt(pos)->ChildList.GetHeadPosition(); pos1 != NULL; )
            // {
                // if (pDoc->CMaterialList.GetAt(pos)->ChildList.GetAt(pos1) == pSelectedElement)
                // pDoc->CMaterialList.GetAt(pos)->ChildList.RemoveAt(pos1);
                // if (pDoc->CMaterialList.GetAt(pos)->ParentList.GetAt(pos1) == pSelectedElement)
                // pDoc->CMaterialList.GetAt(pos)->ParentList.RemoveAt(pos1);
            // }
            // pDoc->CMaterialList.GetNext(pos);
        //}
        //}
        //for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos != NULL; )
        //{
            // for (POSITION pos1 = pDoc->CEnergyList.GetAt(pos)->ChildList.GetHeadPosition(); pos1 != NULL; )
            // {
                // if (pDoc->CEnergyList.GetAt(pos)->ChildList.GetAt(pos1) == pSelectedElement)
                // pDoc->CEnergyList.GetAt(pos)->ChildList.RemoveAt(pos1);
                // if (pDoc->CEnergyList.GetAt(pos)->ParentList.GetAt(pos1) == pSelectedElement)
                // pDoc->CEnergyList.GetAt(pos)->ParentList.RemoveAt(pos1);
            // }
            // pDoc->CEnergyList.GetNext(pos);
        //}

        //OnDraw(this->GetDC());
    }

//
=====
//
=====
// FUNCTIONS FOR DERIVATIONAL TOPOLOGICAL CONSERVATION CHECKS
//
=====
=====

```

```

//
=====
void CConModView::Set_OrphanFlowMsg()
{
    CConModDoc* pDoc = GetDocument();

    Msg_OrphanFlow = "";

    CString* pEdgeNames = new CString;
    *pEdgeNames = _T("");

    for (POSITION pos = pDoc->CMaterialList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CMaterialList.GetAt(pos)->ParentList.IsEmpty() &&
!ElementIsEnv(pDoc->CMaterialList.GetAt(pos)->pTailElem)
            *pEdgeNames = *pEdgeNames + _T(", ") + pDoc-
>CMaterialList.GetAt(pos)->GivenName;

        pDoc->CMaterialList.GetNext(pos);
    }

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CEnergyList.GetAt(pos)->ParentList.IsEmpty() &&
!ElementIsEnv(pDoc->CEnergyList.GetAt(pos)->pTailElem) && !(pDoc-
>CEnergyList.GetAt(pos)->ThisFlowIsIncomingBaggage)
            *pEdgeNames = *pEdgeNames + _T(", ") + pDoc-
>CEnergyList.GetAt(pos)->GivenName;

        pDoc->CEnergyList.GetNext(pos);
    }

    if (*pEdgeNames != "")
        Msg_OrphanFlow = _T("\nOrphan Flow Detected: ") +
*pEdgeNames + (".");

    delete pEdgeNames;
}

void CConModView::Set_BarrenFlowMsg()
{
    CConModDoc* pDoc = GetDocument();

    Msg_BarrenFlow = "";

    CString* pEdgeNames = new CString;
    *pEdgeNames = _T("");
}

```

```

    for (POSITION pos = pDoc->CMaterialList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CMaterialList.GetAt(pos)->ChildList.IsEmpty() &&
!ElementIsEnv(pDoc->CMaterialList.GetAt(pos)->pHeadElem)
            *pEdgeNames = *pEdgeNames + _T(", ") + pDoc-
>CMaterialList.GetAt(pos)->GivenName;

            pDoc->CMaterialList.GetNext(pos);
    }

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        if (pDoc->CEnergyList.GetAt(pos)->ChildList.IsEmpty() &&
!ElementIsEnv(pDoc->CEnergyList.GetAt(pos)->pHeadElem) && !(pDoc-
>CEnergyList.GetAt(pos)->ThisFlowIsOutgoingBaggage))
            *pEdgeNames = *pEdgeNames + _T(", ") + pDoc-
>CEnergyList.GetAt(pos)->GivenName;

            pDoc->CEnergyList.GetNext(pos);
    }

    if (*pEdgeNames != "")
        Msg_BarrenFlow = Msg_BarrenFlow + _T("\nBarren Flow
Detected: ") + *pEdgeNames + (".");

    delete pEdgeNames;
}

void CConModView::Set_OneInManyOutMsg_M()
{
    CConModDoc* pDoc = GetDocument();

    Msg_OneInManyOut_M = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        CString* pInputEdgeName = new CString;
        *pInputEdgeName = _T("");
        CString *pOutputEdgeNames = new CString;
        *pOutputEdgeNames = _T("");

        //=====
// Inference of MATERIAL conservation - One In Many Out
//=====

        for (POSITION pos1 = pDoc->CMaterialList.GetHeadPosition();
pos1 != NULL; )

```



```

        {
            if (pDoc->CMaterialList.GetAt (pos1) ->pHeadElem ==
pDoc->CFunctionList.GetAt (pos) )
                pDoc->CMaterialList_IN_TEMP.AddTail (pDoc-
>CMaterialList.GetAt (pos1));

                pDoc->CMaterialList.GetNext (pos1);
        }

        if (pDoc->CMaterialList_IN_TEMP.GetCount() == 1)
        {
            for (POSITION pos2 = pDoc-
>CMaterialList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CMaterialList.GetAt (pos2) ->pTailElem
== pDoc->CFunctionList.GetAt (pos) )
                    pDoc-
>CMaterialList_OUT_TEMP.AddTail (pDoc->CMaterialList.GetAt (pos2));

                    pDoc->CMaterialList.GetNext (pos2);
            }

            if (pDoc->CMaterialList_OUT_TEMP.GetCount() >= 1)
            {
                *pInputEdgeName = pDoc-
>CMaterialList_IN_TEMP.GetHead() ->GivenName;

                for (POSITION pos3 = pDoc-
>CMaterialList_OUT_TEMP.GetHeadPosition(); pos3 != NULL; )
                {
                    pDoc->CMaterialList_IN_TEMP.GetHead() -
>ChildList.AddTail (pDoc->CMaterialList_OUT_TEMP.GetAt (pos3));
                    pDoc->CMaterialList_OUT_TEMP.GetAt (pos3) -
>ParentList.AddTail (pDoc->CMaterialList_IN_TEMP.GetHead());
                    *pOutputEdgeNames = *pOutputEdgeNames +
_T(", ") + pDoc->CMaterialList_OUT_TEMP.GetAt (pos3) ->GivenName;

                    pDoc-
>CMaterialList_OUT_TEMP.GetNext (pos3);
                }

                Msg_OneInManyOut_M = Msg_OneInManyOut_M +
("\nInferred Derivations: {" + *pInputEdgeName + ("} --> {" +
*pOutputEdgeNames + ("}.");
            }
        }

        delete pInputEdgeName;
        delete pOutputEdgeNames;
        EmptyAllTempLists(); // For every function block

        pDoc->CFunctionList.GetNext (pos);
    }

```

```

}

void CConModView::Set_OneInManyOutMsg_E()
{
    CConModDoc* pDoc = GetDocument();

    Msg_OneInManyOut_E = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        CString* pInputEdgeName = new CString;
        *pInputEdgeName = _T("");
        CString *pOutputEdgeNames = new CString;
        *pOutputEdgeNames = _T("");

        //=====
        // Inference of MATERIAL conservation - One In Many Out
        //=====
        for (POSITION pos1 = pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                pDoc->CEnergyList_IN_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos1));

            pDoc->CEnergyList.GetNext(pos1);
        }

        if (pDoc->CEnergyList_IN_TEMP.GetCount() == 1)
        {
            for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CEnergyList.GetAt(pos2)->pTailElem ==
pDoc->CFunctionList.GetAt(pos))
                    pDoc->CEnergyList_OUT_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos2));

                pDoc->CEnergyList.GetNext(pos2);
            }

            if (pDoc->CEnergyList_OUT_TEMP.GetCount() >= 1)
            {
                *pInputEdgeName = pDoc-
>CEnergyList_IN_TEMP.GetHead()->GivenName;

```

```

        for (POSITION pos3 = pDoc->CEnergyList_OUT_TEMP.GetHeadPosition(); pos3 != NULL; )
        {
            pDoc->CEnergyList_IN_TEMP.GetHead() -
>ChildList.AddTail (pDoc->CEnergyList_OUT_TEMP.GetAt (pos3));
            pDoc->CEnergyList_OUT_TEMP.GetAt (pos3) -
>ParentList.AddTail (pDoc->CEnergyList_IN_TEMP.GetHead());
            *pOutputEdgeNames = *pOutputEdgeNames +
_T(", ") + pDoc->CEnergyList_OUT_TEMP.GetAt (pos3)->GivenName;

            pDoc->CEnergyList_OUT_TEMP.GetNext (pos3);
        }

        Msg_OneInManyOut_E = Msg_OneInManyOut_E +
("\nInferred Derivations: {" + *pInputEdgeName + ("} --> {" +
*pOutputEdgeNames + ("}.");
    }
}

delete pInputEdgeName;
delete pOutputEdgeNames;
EmptyAllTempLists(); // For every function block

pDoc->CFunctionList.GetNext (pos);
}
}

void CConModView::Set_ManyInOneOutMsg_M()
{
    CConModDoc* pDoc = GetDocument();

    Msg_ManyInOneOut_M = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        CString *pInputEdgeNames = new CString;
        *pInputEdgeNames = _T("");
        CString *pOutputEdgeName = new CString;
        *pOutputEdgeName = _T("");

        //=====
        // Inference of MATERIAL conservation - Many In One Out
        //=====

        for (POSITION pos1 = pDoc->CMaterialList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CMaterialList.GetAt (pos1)->pHeadElem ==
pDoc->CFunctionList.GetAt (pos))

```

```

        pDoc->CMaterialList_IN_TEMP.AddTail(pDoc-
>CMaterialList.GetAt(pos1));

        pDoc->CMaterialList.GetNext(pos1);
    }

    if (pDoc->CMaterialList_IN_TEMP.GetCount() > 1)
    {
        for (POSITION pos2 = pDoc-
>CMaterialList.GetHeadPosition(); pos2 != NULL; )
        {
            if (pDoc->CMaterialList.GetAt(pos2)->pTailElem
== pDoc->CFunctionList.GetAt(pos))
                pDoc-
>CMaterialList_OUT_TEMP.AddTail(pDoc->CMaterialList.GetAt(pos2));

            pDoc->CMaterialList.GetNext(pos2);
        }

        if (pDoc->CMaterialList_OUT_TEMP.GetCount() == 1)
        {
            *pOutputEdgeName = pDoc-
>CMaterialList_OUT_TEMP.GetHead()->GivenName;

            for (POSITION pos3 = pDoc-
>CMaterialList_IN_TEMP.GetHeadPosition(); pos3 != NULL; )
            {
                pDoc->CMaterialList_OUT_TEMP.GetHead()-
>ParentList.AddTail(pDoc->CMaterialList_IN_TEMP.GetAt(pos3));
                pDoc->CMaterialList_IN_TEMP.GetAt(pos3)-
>ChildList.AddTail(pDoc->CMaterialList_OUT_TEMP.GetHead());
                *pInputEdgeNames = *pInputEdgeNames +
_T(", ") + pDoc->CMaterialList_IN_TEMP.GetAt(pos3)->GivenName;

                pDoc-
>CMaterialList_IN_TEMP.GetNext(pos3);
            }

            Msg_ManyInOneOut_M = Msg_ManyInOneOut_M +
("\nInferred Derivations: {" + *pInputEdgeNames + ("} --> {" +
*pOutputEdgeName + ("}.");
        }
    }
    delete pInputEdgeNames;
    delete pOutputEdgeName;
    EmptyAllTempLists(); // For every function block
    pDoc->CFunctionList.GetNext(pos);
}

void CConModView::Set_ManyInOneOutMsg_E()
{
    CConModDoc* pDoc = GetDocument();

```

```

    Msg_ManyInOneOut_E = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        CString *pInputEdgeNames = new CString;
        *pInputEdgeNames = _T("");
        CString *pOutputEdgeName = new CString;
        *pOutputEdgeName = _T("");

        //=====
        // Inference of MATERIAL conservation - Many In One Out
        //=====

        for (POSITION pos1 = pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                pDoc->CEnergyList_IN_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos1));

                pDoc->CEnergyList.GetNext(pos1);
        }

        if (pDoc->CEnergyList_IN_TEMP.GetCount() > 1)
        {
            for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CEnergyList.GetAt(pos2)->pTailElem ==
pDoc->CFunctionList.GetAt(pos))
                    pDoc->CEnergyList_OUT_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos2));

                    pDoc->CEnergyList.GetNext(pos2);
            }

            if (pDoc->CEnergyList_OUT_TEMP.GetCount() == 1)
            {
                *pOutputEdgeName = pDoc-
>CEnergyList_OUT_TEMP.GetHead()->GivenName;

                for (POSITION pos3 = pDoc-
>CEnergyList_IN_TEMP.GetHeadPosition(); pos3 != NULL; )
                {
                    pDoc->CEnergyList_OUT_TEMP.GetHead()-
>ParentList.AddTail(pDoc->CEnergyList_IN_TEMP.GetAt(pos3));

```

```

        pDoc->CEnergyList_IN_TEMP.GetAt(pos3)-
>ChildList.AddTail(pDoc->CEnergyList_OUT_TEMP.GetHead());
        *pInputEdgeNames = *pInputEdgeNames +
_T(", ") + pDoc->CEnergyList_IN_TEMP.GetAt(pos3)->GivenName;

        pDoc->CEnergyList_IN_TEMP.GetNext(pos3);
    }

    Msg_ManyInOneOut_M = Msg_ManyInOneOut_M +
("\nInferred Derivations: {") + *pInputEdgeNames + ("} --> {") +
*pOutputEdgeName + ("}.");
    }
}
delete pInputEdgeNames;
delete pOutputEdgeName;
EmptyAllTempLists(); // For every function block
pDoc->CFunctionList.GetNext(pos);
}
}

void CConModView::Set_ManyInManyOutMsg()
{
    CConModDoc* pDoc = GetDocument();

    Msg_ManyInManyOut = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        CString *pInputEdgeNames = new CString;
        *pInputEdgeNames = _T("");
        CString *pOutputEdgeNames = new CString;
        *pOutputEdgeNames = _T("");

        //=====
        // Inference of Impossible Conclusion - Many In Many Out
        //=====
        for (POSITION pos1 = pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                pDoc->CEnergyList_IN_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos1));

            pDoc->CEnergyList.GetNext(pos1);
        }
    }
}

```

```

        if (pDoc->CEnergyList_IN_TEMP.GetCount() > 1) // Only
then you investigate further, otherwise don't waste time
        {
            for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CEnergyList.GetAt(pos2)->pTailElem ==
pDoc->CFunctionList.GetAt(pos))
                    pDoc->CEnergyList_OUT_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos2));

                pDoc->CEnergyList.GetNext(pos2);
            }

            if (pDoc->CEnergyList_OUT_TEMP.GetCount() > 1) //
Now both sides have too many flows to conclude
            {
                for (POSITION pos3 = pDoc-
>CEnergyList_IN_TEMP.GetHeadPosition(); pos3 != NULL; )
                {
                    for (POSITION pos4 = pDoc-
>CEnergyList_OUT_TEMP.GetHeadPosition(); pos4 != NULL; )
                    {
                        pDoc-
>CEnergyList_OUT_TEMP.GetAt(pos4)->ParentList.AddTail(pDoc-
>CEnergyList_IN_TEMP.GetAt(pos3));
                        pDoc-
>CEnergyList_IN_TEMP.GetAt(pos3)->ChildList.AddTail(pDoc-
>CEnergyList_OUT_TEMP.GetAt(pos4));
                        pDoc-
>CEnergyList_OUT_TEMP.GetNext(pos4);
                    }
                    *pInputEdgeNames = *pInputEdgeNames +
_T(", ") + pDoc->CEnergyList_IN_TEMP.GetAt(pos3)->GivenName;

                    pDoc->CEnergyList_IN_TEMP.GetNext(pos3);
                }/**/

                for (POSITION pos5 = pDoc-
>CEnergyList_OUT_TEMP.GetHeadPosition(); pos5 != NULL; )
                {
                    *pOutputEdgeNames = *pOutputEdgeNames +
_T(", ") + pDoc->CEnergyList_OUT_TEMP.GetAt(pos5)->GivenName;
                    pDoc->CEnergyList_OUT_TEMP.GetNext(pos5);
                }

                Msg_ManyInManyOut = Msg_ManyInManyOut +
("\nInferred Derivations: {" + *pInputEdgeNames + ("} --> {" +
*pOutputEdgeNames + ("}.");
            }
        }

        *pInputEdgeNames = _T("");

```

```

        *pOutputEdgeNames = _T("");
        EmptyAllTempLists(); // For every function block

        for (POSITION pos1 = pDoc->CMaterialList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CMaterialList.GetAt(pos1)->pHeadElem ==
pDoc->CFunctionList.GetAt(pos))
                pDoc->CMaterialList_IN_TEMP.AddTail(pDoc-
>CMaterialList.GetAt(pos1));

                pDoc->CMaterialList.GetNext(pos1);
        }

        if (pDoc->CMaterialList_IN_TEMP.GetCount() > 1) // Only
then you investigate further, otherwise don't waste time
        {
            for (POSITION pos2 = pDoc-
>CMaterialList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CMaterialList.GetAt(pos2)->pTailElem
== pDoc->CFunctionList.GetAt(pos))
                    pDoc-
>CMaterialList_OUT_TEMP.AddTail(pDoc->CMaterialList.GetAt(pos2));

                    pDoc->CMaterialList.GetNext(pos2);
            }

            if (pDoc->CMaterialList_OUT_TEMP.GetCount() > 1)
// Now both sides have too many flows to conclude
            {
                for (POSITION pos3 = pDoc-
>CMaterialList_IN_TEMP.GetHeadPosition(); pos3 != NULL; )
                {
                    for (POSITION pos4 = pDoc-
>CMaterialList_OUT_TEMP.GetHeadPosition(); pos4 != NULL; )
                    {
                        pDoc-
>CMaterialList_OUT_TEMP.GetAt(pos4)->ParentList.AddTail(pDoc-
>CMaterialList_IN_TEMP.GetAt(pos3));
                        pDoc-
>CMaterialList_IN_TEMP.GetAt(pos3)->ChildList.AddTail(pDoc-
>CMaterialList_OUT_TEMP.GetAt(pos4));
                        pDoc-
>CMaterialList_OUT_TEMP.GetNext(pos4);
                    }
                    *pInputEdgeNames = *pInputEdgeNames +
_T(", ") + pDoc->CMaterialList_IN_TEMP.GetAt(pos3)->GivenName;

                    pDoc-
>CMaterialList_IN_TEMP.GetNext(pos3);
                }/**/
            }
        }

```



```

        for (POSITION pos5 = pDoc-
>CMaterialList_OUT_TEMP.GetHeadPosition(); pos5 != NULL; )
        {
            *pOutputEdgeNames = *pOutputEdgeNames +
_T(", ") + pDoc->CMaterialList_OUT_TEMP.GetAt(pos5)->GivenName;
            pDoc-
>CMaterialList_OUT_TEMP.GetNext(pos5);
        }

        Msg_ManyInManyOut = Msg_ManyInManyOut +
("\nInferred Derivations: {") + *pInputEdgeNames + ("} --> {") +
*pOutputEdgeNames + ("}.");
    }

    delete pInputEdgeNames;
    delete pOutputEdgeNames;
    EmptyAllTempLists(); // For every function block
    pDoc->CFunctionList.GetNext(pos);
}

void CConModView::Set_MissingResidualEnergyMsg()
{
    if (ReasoningOption == QUALITATIVE_CONSERVATION)
        return;

    CConModDoc* pDoc = GetDocument();

    Msg_MissingResidualEnergy = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        for (POSITION pos1 = pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                pDoc->CEnergyList_IN_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos1));

            pDoc->CEnergyList.GetNext(pos1);
        }

        if (pDoc->CEnergyList_IN_TEMP.GetCount() >= 1)
        {
            for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
            {
                if (pDoc->CEnergyList.GetAt(pos2)->pTailElem ==
pDoc->CFunctionList.GetAt(pos))

```

```

        pDoc->CEnergyList_OUT_TEMP.AddTail(pDoc-
>CEnergyList.GetAt(pos2));

        pDoc->CEnergyList.GetNext(pos2);
    }

    if (pDoc->CEnergyList_OUT_TEMP.GetCount() >= 1)
    {
        // Testing if there is at least one residual
energy flow at the output
        bool ResidualEnergyFound = false;

        for (POSITION pos3 = pDoc-
>CEnergyList_OUT_TEMP.GetHeadPosition(); pos3 != NULL; )
        {
            if (pDoc-
>CEnergyList_OUT_TEMP.GetAt(pos3)->IsResidual)
                ResidualEnergyFound = true;
            pDoc->CEnergyList_OUT_TEMP.GetNext(pos3);
        }

        if (!ResidualEnergyFound)
        {
            Msg_MissingResidualEnergy =
Msg_MissingResidualEnergy +
                "\n::Warning:: Energy Loss Not
Shown in Function: " + pDoc->CFunctionList.GetAt(pos)->GivenName + ".";
        }
    }

    EmptyAllTempLists(); // For every function block
    pDoc->CFunctionList.GetNext(pos);
}

void CConModView::Set_MaterialChangeWithoutEnergyMsg()
{
    CConModDoc* pDoc = GetDocument();

    Msg_MaterialChangeWithoutEnergy = "";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        bool ThisFuncHasInputM;
        bool ThisFuncHasOutputM;
        bool ThisFuncHasInputEBaggage;
        bool ThisFuncHasOutputEBaggage;

        ThisFuncHasInputM = false;
        ThisFuncHasOutputM = false;
        ThisFuncHasInputEBaggage = false;
    }
}

```

```

        ThisFuncHasOutputEBaggage = false;

        for (POSITION pos1 = pDoc->CMaterialList.GetHeadPosition();
pos1 != NULL; )
        {
            if (pDoc->CMaterialList.GetAt(pos1)->pHeadElem ==
pDoc->CFunctionList.GetAt(pos))
            {
                ThisFuncHasInputM = true;

                for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
                {
                    if ((pDoc->CEnergyList.GetAt(pos2)-
>pTailElem == pDoc->CMaterialList.GetAt(pos1)) &&
(pDoc->CEnergyList.GetAt(pos2)-
>pHeadElem == pDoc->CFunctionList.GetAt(pos)))
                    {
                        ThisFuncHasInputEBaggage = true;
                        //return;
                    }
                    pDoc->CEnergyList.GetNext(pos2);
                }
            }

            if (pDoc->CMaterialList.GetAt(pos1)->pTailElem ==
pDoc->CFunctionList.GetAt(pos))
            {
                ThisFuncHasOutputM = true;

                for (POSITION pos2 = pDoc-
>CEnergyList.GetHeadPosition(); pos2 != NULL; )
                {
                    if ((pDoc->CEnergyList.GetAt(pos2)-
>pHeadElem == pDoc->CMaterialList.GetAt(pos1)) &&
(pDoc->CEnergyList.GetAt(pos2)-
>pTailElem == pDoc->CFunctionList.GetAt(pos)))
                    {
                        ThisFuncHasOutputEBaggage = true;
                        //return;
                    }
                    pDoc->CEnergyList.GetNext(pos2);
                }
            }

            pDoc->CMaterialList.GetNext(pos1);
        }

        if ((ThisFuncHasInputM) && (ThisFuncHasOutputM) &&
!(ThisFuncHasInputEBaggage) &&!(ThisFuncHasOutputEBaggage))
            Msg_MaterialChangeWithoutEnergy =
Msg_MaterialChangeWithoutEnergy +

```

```

        "\nEnergy must be exchanged to/from Material to
transform Material (" + pDoc->CFunctionList.GetAt(pos)->GivenName +
").";

        EmptyAllTempLists(); // For every function block
        pDoc->CFunctionList.GetNext(pos);
    }
}/**/

void CConModView::EmptyAllTempLists()
{
    CConModDoc* pDoc = GetDocument();

    pDoc->CMaterialList_IN_TEMP.RemoveAll();
    pDoc->CMaterialList_OUT_TEMP.RemoveAll();
    pDoc->CEnergyList_IN_TEMP.RemoveAll();
    pDoc->CEnergyList_OUT_TEMP.RemoveAll();
    pDoc->CSignalList_IN_TEMP.RemoveAll();
    pDoc->CSignalList_OUT_TEMP.RemoveAll();
}

void CConModView::ComposeQualitativeMessage()
{
    CConModDoc* pDoc = this->GetDocument();

    //=====
    // Derivation Check
    //=====

    Msg_OneInManyOut_M = "";
    Msg_OneInManyOut_E = "";
    Msg_ManyInOneOut_M = "";
    Msg_ManyInOneOut_E = "";
    Msg_ManyInManyOut = "";
    Msg_MissingResidualEnergy = "";
    Msg_MaterialChangeWithoutEnergy = "";
    Msg_OrphanFlow = "";
    Msg_BarrenFlow = "";

    CString* pMsg_DerivationChecks = new CString;
    *pMsg_DerivationChecks = _T("***** QUALITATIVE CONSERVATION
REPORT *****\n");

    // First, clear all existing parent-child relations that are
    // leftover from a previous call to this function.
    // The relations will be recomputed during the next inferences
anyways.
    for (POSITION pos = pDoc->CMaterialList.GetHeadPosition(); pos !=
NULL; )
    {
        pDoc->CMaterialList.GetAt(pos)->ParentList.RemoveAll();
        pDoc->CMaterialList.GetAt(pos)->ChildList.RemoveAll();
        pDoc->CMaterialList.GetNext(pos);
    }
}

```

```

    }

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        pDoc->CEnergyList.GetAt(pos)->ParentList.RemoveAll();
        pDoc->CEnergyList.GetAt(pos)->ChildList.RemoveAll();
        pDoc->CEnergyList.GetNext(pos);
    }

    // Must finish drawing inferences before deciding barren and
orphan flows,
    // because it is during these inferences that parent and children
are
    // computed. Without these inferences, all flows will return as
both
    // orphan abd barren.
    Set_OneInManyOutMsg_M();
    Set_OneInManyOutMsg_E();
    Set_ManyInOneOutMsg_M();
    Set_ManyInOneOutMsg_E();
    Set_ManyInManyOutMsg();
    Set_MissingResidualEnergyMsg();
    Set_MaterialChangeWithoutEnergyMsg();

    // Now call orphan and barren flow messages
    Set_OrphanFlowMsg();
    Set_BarrenFlowMsg();

    // Now compose all the messages generated by the above checks and
display
    *pMsg_DerivationChecks = *pMsg_DerivationChecks +
                                Msg_OneInManyOut_M +
                                Msg_OneInManyOut_E +
                                Msg_ManyInOneOut_M +
                                Msg_ManyInOneOut_E +
                                Msg_ManyInManyOut +

    Msg_MaterialChangeWithoutEnergy +
                                Msg_OrphanFlow +
                                Msg_BarrenFlow;

    int* m = new int;
    *m      =      MessageBox(*pMsg_DerivationChecks,      _T("Qualitative
Conservation Report"), MB_ICONWARNING | MB_OK);
    delete m;

    delete pMsg_DerivationChecks;      // Resets to empty string

    //=====
    // Irreversibility Check
    //=====

```

```

    CString* pMsg_IrrevChecks = new CString;
    *pMsg_IrrevChecks = _T("***** QUALITATIVE IRREVERSIBILITY REPORT
*****\n");

    if (ReasoningOption >= QUALITATIVE_IRREVERSIBILITY)
    {
        *pMsg_IrrevChecks = *pMsg_IrrevChecks +
Msg_MissingResidualEnergy;
        int* n = new int;
        *n = MessageBox(*pMsg_IrrevChecks, _T("Qualitative
Irreversibility Report"), MB_ICONWARNING | MB_OK);
        delete n;
    }

    delete pMsg_IrrevChecks;
}

void CConModView::VerifyPositivePowerOfFlows()
{
    CConModDoc* pDoc = this->GetDocument();

    ContinueReasoning = true;

    CString* pNegativeEnergyReportString = new CString;
    *pNegativeEnergyReportString = "***** NEGATIVE POWER REPORT
*****\nThe following flows have negative power. \n";

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        pDoc->CEnergyList.GetAt(pos)->Power = pDoc-
>CEnergyList.GetAt(pos)->UI_ForceTerm *
        pDoc->CEnergyList.GetAt(pos)->UI_RateTerm;

        if (pDoc->CEnergyList.GetAt(pos)->Power < 0)
        {
            ContinueReasoning = false;
            CString* pPowerString = new CString;
            pPowerString->Format(_T("%.1f"), pDoc-
>CEnergyList.GetAt(pos)->Power);

            *pNegativeEnergyReportString =
*pNegativeEnergyReportString + "\nFlow: " + pDoc-
>CEnergyList.GetAt(pos)->GivenName +
            "\t\tPower = " + *pPowerString + " W";

            delete pPowerString;
        }

        pDoc->CEnergyList.GetNext(pos);
    }

    if (ContinueReasoning == false)

```

```

    {
        int n = MessageBox(*pNegativeEnergyReportString,
_T("Negative Power Report"), MB_ICONWARNING | MB_OK);
        AfxMessageBox(_T("Quantitative reasoning (Energy Balance,
Efficiency, Confluence) cannot continue with flows with negative
power."));
    }

    delete pNegativeEnergyReportString;
}

void CConModView::VerifyEnergyBalanceOfFunctions()
{
    if (ContinueReasoning == false)
        return;

    CConModDoc* pDoc = this->GetDocument();

    CString* pEnergyBalanceReportString = new CString;
    *pEnergyBalanceReportString = "***** ENERGY BALANCE REPORT
*****\n";

    for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
    {
        double* pTotalInputPower = new double;
        double* pTotalOutputPower = new double;

        *pTotalInputPower = 0.0;
        *pTotalOutputPower = 0.0;

        for(POSITION pos1 = pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
        {
            pDoc->CEnergyList.GetAt(pos1)->Power = pDoc-
>CEnergyList.GetAt(pos1)->UI_ForceTerm *
            pDoc->CEnergyList.GetAt(pos1)->UI_RateTerm;

            if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                *pTotalInputPower = *pTotalInputPower + (pDoc-
>CEnergyList.GetAt(pos1)->Power);

            if (pDoc->CEnergyList.GetAt(pos1)->pTailElem == pDoc-
>CFunctionList.GetAt(pos))
                *pTotalOutputPower = *pTotalOutputPower +
(pDoc->CEnergyList.GetAt(pos1)->Power);

            pDoc->CEnergyList.GetNext(pos1);
        }

        if (*pTotalInputPower == *pTotalOutputPower)

```

```

        *pEnergyBalanceReportString =
*pEnergyBalanceReportString + "\nFunction: " + pDoc-
>CFunctionList.GetAt(pos)->GivenName + "\tBalanced.";

        else
        {
            ContinueReasoning = false;

            CString* pInputPString = new CString;
            CString* pOutputPString = new CString;

            pInputPString->Format(_T("%4.1f"),
*pTotalInputPower);
            pOutputPString->Format(_T("%4.1f"),
*pTotalOutputPower);

            *pEnergyBalanceReportString =
*pEnergyBalanceReportString + "\nFunction: " +
                pDoc->CFunctionList.GetAt(pos)->GivenName +
"\tInput = " + *pInputPString +
                " W\tOutput = " + *pOutputPString + " W.";

            delete pInputPString;
            delete pOutputPString;
        }

        delete pTotalInputPower;
        delete pTotalOutputPower;

        pDoc->CFunctionList.GetNext(pos);
    }

    int n = MessageBox(*pEnergyBalanceReportString, _T("Energy
Balance Violation Report"), MB_ICONWARNING | MB_OK);

    if (ContinueReasoning == false)
        AfxMessageBox(_T("Quantitative reasoning (Efficiency,
Confluence) cannot continue without energy balance in each
function."));

    delete pEnergyBalanceReportString;
}

void CConModView::ComputeEfficiency()
{
    if (ContinueReasoning == false)
        return;

    CConModDoc* pDoc = this->GetDocument();

    //=====
    // Compute function-wise efficiency
    //=====

```



```

        CString* pEfficiencyMessage = new CString;
        *pEfficiencyMessage = "*****  INDIVIDUAL  FUNCTION  EFFICIENCY
REPORT *****\n"
        "\nFunction\tInput\tUsable\tLoss\tEfficiency"
        "\n=====";

        for (POSITION pos = pDoc->CFunctionList.GetHeadPosition(); pos !=
NULL; )
        {
            pDoc->CFunctionList.GetAt(pos)->Efficiency = 0.0;
            // Reset the efficiency at the beginning of
            // each run of this algorithm

            double* pTotalInputPower = new double;
            double* pTotalUsableOutputPower = new double;

            *pTotalInputPower = 0.0;
            *pTotalUsableOutputPower = 0.0;

            for(POSITION  pos1  =  pDoc->CEnergyList.GetHeadPosition();
pos1 != NULL; )
            {
                pDoc->CEnergyList.GetAt(pos1)->Power      =      pDoc-
>CEnergyList.GetAt(pos1)->UI_ForceTerm *
                pDoc->CEnergyList.GetAt(pos1)->UI_RateTerm;

                if (pDoc->CEnergyList.GetAt(pos1)->pHeadElem == pDoc-
>CFunctionList.GetAt(pos))
                    *pTotalInputPower = *pTotalInputPower + (pDoc-
>CEnergyList.GetAt(pos1)->Power);

                if      ((pDoc->CEnergyList.GetAt(pos1)->pTailElem      ==
pDoc->CFunctionList.GetAt(pos)) &&
                        (pDoc->CEnergyList.GetAt(pos1)->IsResidual      ==
false))
                    *pTotalUsableOutputPower      =
*pTotalUsableOutputPower + (pDoc->CEnergyList.GetAt(pos1)->Power);

                pDoc->CEnergyList.GetNext(pos1);
            }

            if ((*pTotalInputPower != 0.0) && (*pTotalUsableOutputPower
!= 0))
                pDoc->CFunctionList.GetAt(pos)->Efficiency      =
(*pTotalUsableOutputPower / *pTotalInputPower);

            CString* pInputEString = new CString;
            CString* pUsableOutputEString = new CString;
            CString* pLossEString = new CString;
            CString* pEffyString = new CString;

            pInputEString->Format(_T("%5.1f"), *pTotalInputPower);

```

```

        pUsableOutputEString->Format(_T("%5.1f"),
*pTotalUsableOutputPower);
        pLossEString->Format(_T("%5.1f"),      (*pTotalInputPower -
*pTotalUsableOutputPower));
        pEffyString->Format(_T("%5.3f"),      pDoc-
>CFunctionList.GetAt(pos)->Efficiency);

        *pEfficiencyMessage = *pEfficiencyMessage +
            "\n" + pDoc->CFunctionList.GetAt(pos)->GivenName +
            "\t" + *pInputEString +
            "\t" + *pUsableOutputEString +
            "\t" + *pLossEString +
            "\t" + *pEffyString;

        delete pTotalInputPower;
        delete pTotalUsableOutputPower;
        delete pInputEString;
        delete pUsableOutputEString;
        delete pLossEString;
        delete pEffyString;

        pDoc->CFunctionList.GetNext(pos);
    }

    //=====
    // Compute efficiency for the whole model
    //=====

    double* pModelInputPower = new double;
    double* pModelLossPower = new double;
    double* pModelEfficiency = new double;

    *pModelInputPower = 0;
    *pModelLossPower = 0;
    *pModelEfficiency = 0;

    for (POSITION pos = pDoc->CEnergyList.GetHeadPosition(); pos !=
NULL; )
    {
        if      ((ElementIsFunction(pDoc->CEnergyList.GetAt(pos)-
>pHeadElem)) &&
                (pDoc->CEnergyList.GetAt(pos)->pTailElem != NULL) &&
                ((ElementIsEnv(pDoc->CEnergyList.GetAt(pos)-
>pTailElem)) || (ElementIsEnv(pDoc->CEnergyList.GetAt(pos)->pTailElem-
>pTailElem))))
            *pModelInputPower = *pModelInputPower + pDoc-
>CEnergyList.GetAt(pos)->Power;

        if      ((ElementIsFunction(pDoc->CEnergyList.GetAt(pos)-
>pTailElem)) &&
                (pDoc->CEnergyList.GetAt(pos)->pHeadElem != NULL) &&

```

```

        ((ElementIsEnv (pDoc->CEnergyList.GetAt (pos) -
>pHeadElem)) || (ElementIsEnv (pDoc->CEnergyList.GetAt (pos) ->pHeadElem-
>pHeadElem))) &&
        (pDoc->CEnergyList.GetAt (pos) ->IsResidual == true))
        *pModelLossPower = *pModelLossPower + pDoc-
>CEnergyList.GetAt (pos) ->Power;

        pDoc->CEnergyList.GetNext (pos);
    }

    if ((*pModelInputPower != 0) /*&& (*pModelLossPower != 0)*/)
    // Have to set more traps
        *pModelEfficiency = (*pModelInputPower - *pModelLossPower)
/ *pModelInputPower;

    CString* pModelEffyString = new CString;
    pModelEffyString->Format(_T("%5.3f"), *pModelEfficiency);

    *pEfficiencyMessage = *pEfficiencyMessage + "\n\nOVERAL MODEL
EFFICIENCY: " + *pModelEffyString;

    delete pModelInputPower;
    delete pModelLossPower;
    delete pModelEfficiency;
    delete pModelEffyString;

    int n = MessageBox(*pEfficiencyMessage, _T("Efficiency Report"),
MB_ICONWARNING | MB_OK);

    delete pEfficiencyMessage;
}

void CConModView::ComposeQuantitativeMessage()
{
    //=====
    // Commented out for rolling back to Layer One (Chapter 6)
    //=====

    if ((ReasoningOption == QUALITATIVE_CONSERVATION) ||
(ReasoningOption == QUALITATIVE_IRREVERSIBILITY))
    {
        AfxMessageBox(_T("***** QUANTITATIVE REASONING NOT
AVAILABLE *****\n\nTo turn on, choose \"Quantitative -> Efficiency\"
from Reasoning Menu."));
        return;
    }

    if (ReasoningOption == QUANTITATIVE_EFFICIENCY)
    {
        VerifyPositivePowerOfFlows();
        VerifyEnergyBalanceOfFunctions();
        ComputeEfficiency(); // Resets every function's effy to
zero, then recomputes

```

```

                                                                    // from present
state of model
}

if (ReasoningOption == QUANTITATIVE_POWERREQUIRED)
{
    AfxMessageBox(_T("Under Construction.));
}
}

```

```

// Convert_E.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Convert_E_Template.h"

// CConvert_E dialog

IMPLEMENT_DYNAMIC(CConvert_E_Template, CDialog)

CConvert_E_Template::CConvert_E_Template(CWnd* pParent /*= NULL*/,
CPoint InsertionPoint /*= (500,500)*/,
CString* pCounterString_F /*= NULL*/,
CString* pCounterString_InE /*= NULL*/,
CString* pCounterString_OutE /*=
NULL*/, CString* pCounterString_OutE_Res /*= NULL*/)
: CDialog(CConvert_E_Template::IDD, pParent)
{
    pFunctionBlock = new CFunction(NULL, InsertionPoint,
pCounterString_F);

    CPoint TailOfInE(InsertionPoint.x - TEMPLATE_FLOW_LENGTH,
InsertionPoint.y);
    CPoint HeadOfOutE(InsertionPoint.x + TEMPLATE_FLOW_LENGTH,
InsertionPoint.y);
    CPoint HeadOfOutE_Res(InsertionPoint.x, InsertionPoint.y +
TEMPLATE_FLOW_LENGTH);

    pEnergy_InE = new CEnergy(NULL, TailOfInE, InsertionPoint,
pCounterString_InE);
    pEnergy_OutE = new CEnergy(NULL, InsertionPoint, HeadOfOutE,
pCounterString_OutE);
    pEnergy_OutE_Res = new CEnergy(NULL, InsertionPoint,
HeadOfOutE_Res, pCounterString_OutE_Res);
}

```

```

    pEnergy_InE->pHeadElem = pFunctionBlock;
    pEnergy_OutE->pTailElem = pFunctionBlock;
    pEnergy_OutE_Res->pTailElem = pFunctionBlock;
    pEnergy_OutE_Res->UI_IsResidual = true;
}

CConvert_E_Template::~CConvert_E_Template()
{
}

void CConvert_E_Template::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CConvert_E_Template, CDialog)
END_MESSAGE_MAP()

// CConvert_E message handlers

```

```

// DeEn_M_Template.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "DeEn_M_Template.h"

// CDeEn_M_Template dialog

IMPLEMENT_DYNAMIC(CDeEn_M_Template, CDialog)

CDeEn_M_Template::CDeEn_M_Template(CWnd* pParent /*= NULL*/,
    CPoint InsertionPoint /*= (500,500)*/,
    CString* pCounterString_F /*= NULL*/,
    CString* pCounterString_InM /*= NULL*/,
    CString* pCounterString_OutM /*= NULL*/,
    CString* pCounterString_InE /*= NULL*/,
    CString* pCounterString_OutE /*= NULL*/)
    : CDialog(CDeEn_M_Template::IDD, pParent)
{
    pFunctionBlock = new CFunction(NULL, InsertionPoint,
    pCounterString_F);

    CPoint TailOfInM(InsertionPoint.x - 1.5*TEMPLATE_FLOW_LENGTH,
    InsertionPoint.y);
    CPoint HeadOfOutM(InsertionPoint.x + TEMPLATE_FLOW_LENGTH,
    InsertionPoint.y);

```

```

        CPoint      HeadOfOutE(InsertionPoint.x,      InsertionPoint.y      -
TEMPLATE_FLOW_LENGTH);

        pMaterial_InM = new CMaterial(NULL, TailOfInM, InsertionPoint,
pCounterString_InM);
        pMaterial_OutM = new CMaterial(NULL, InsertionPoint, HeadOfOutM,
pCounterString_OutM);
        pEnergy_InE = new CEnergy(NULL, InsertionPoint /*Dummy*/,
InsertionPoint, pCounterString_InE);
        pEnergy_OutE = new CEnergy(NULL, InsertionPoint, HeadOfOutE,
pCounterString_OutE);

        pMaterial_InM->pHeadElem = pFunctionBlock;
        pMaterial_OutM->pTailElem = pFunctionBlock;
        pEnergy_InE->pHeadElem = pFunctionBlock;
        pEnergy_InE->pTailElem = pMaterial_InM;
        pEnergy_OutE->pTailElem = pFunctionBlock;
    }

CDeEn_M_Template::~CDeEn_M_Template()
{
}

void CDeEn_M_Template::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CDeEn_M_Template, CDialog)
END_MESSAGE_MAP()

// CDeEn_M_Template message handlers

```

```

// Distribute_E_Template.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Distribute_E_Template.h"

// CDistribute_E_Template dialog

IMPLEMENT_DYNAMIC(CDistribute_E_Template, CDialog)

CDistribute_E_Template::CDistribute_E_Template(CWnd* pParent /*=
NULL*/,
CPoint InsertionPoint /*= (500,500)*/,

```

```

CString* pCounterString_F /*= NULL*/,
CString* pCounterString_InE /*= NULL*/,
CString* pCounterString_OutE1 /*= NULL*/,
CString* pCounterString_OutE2 /*= NULL*/)
    : CDialog(CDistribute_E_Template::IDD, pParent)
{
    pFunctionBlock = new CFunction(NULL, InsertionPoint,
pCounterString_F);

    CPoint TailOfInE(InsertionPoint.x - TEMPLATE_FLOW_LENGTH,
InsertionPoint.y);
    CPoint HeadOfOutE1(InsertionPoint.x + TEMPLATE_FLOW_LENGTH,
InsertionPoint.y - TEMPLATE_FLOW_LENGTH);
    CPoint HeadOfOutE2(InsertionPoint.x + TEMPLATE_FLOW_LENGTH,
InsertionPoint.y + TEMPLATE_FLOW_LENGTH);

    pEnergy_InE = new CEnergy(NULL, TailOfInE, InsertionPoint,
pCounterString_InE);
    pEnergy_OutE1 = new CEnergy(NULL, InsertionPoint, HeadOfOutE1,
pCounterString_OutE1);
    pEnergy_OutE2 = new CEnergy(NULL, InsertionPoint, HeadOfOutE2,
pCounterString_OutE2);

    pEnergy_InE->pHeadElem = pFunctionBlock;
    pEnergy_OutE1->pTailElem = pFunctionBlock;
    pEnergy_OutE2->pTailElem = pFunctionBlock;
}

CDistribute_E_Template::~CDistribute_E_Template()
{
}

void CDistribute_E_Template::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CDistribute_E_Template, CDialog)
END_MESSAGE_MAP()

// CDistribute_E_Template message handlers

```

```

#include "StdAfx.h"
#include "Edge.h"

```

```

#include "math.h"

CEdge::CEdge(void)
{
}

CEdge::CEdge(CPoint TailClick, CPoint HeadClick)
{
    TailPoint = TailClick;
    HeadPoint = HeadClick;
    GeometricCenter = *InterpolatePoints(TailPoint, HeadPoint, 0.5);
    StemThickness = THIN;
    StemLineFont = PS_SOLID;

    HeadSize = EDGE_HEAD_SIZE;
    HalfHeadAngle = EDGE_HEAD_HALF_ANGLE;

    ComputeAnchorPoints();
    pHeadElem = NULL;
    pTailElem = NULL;
    ThisFlowIsIncomingBaggage = false;
    ThisFlowIsOutgoingBaggage = false;

    FontSize = GENERIC_FONT_SIZE;
}

CEdge::~CEdge(void)
{
}

void CEdge::AttachEdgeToNearestAnchor()
{
    FontSize = GENERIC_FONT_SIZE;
    HeadSize = EDGE_HEAD_SIZE;

    if (pTailElem != NULL)
    {
        // Initialize with any one anchor point - zeroth chosen
        // arbitrarily
        long double d = distance(HeadPoint, pTailElem->Anchors[0]);
        TailPoint = pTailElem->Anchors[0];

        for (int n = 1; n <= 15; n++) // Hard coded - 16 anchor
        // points on both nodes and edges
        {
            if (distance(pTailElem->Anchors[n], HeadPoint) < d)
            {
                d = distance(HeadPoint, pTailElem->Anchors[n]);
                TailPoint = pTailElem->Anchors[n];
            }
        }
    }
}

```



```

    if (pHeadElem != NULL)
    {
        // Initialize with any one anchor point - zeroth chosen
        arbitrarily
        long double d = distance(TailPoint, pHeadElem->Anchors[0]);
        this->HeadPoint = pHeadElem->Anchors[0];

        for (int n = 1; n <= 15; n++) // Hard coded - 16 anchor
        points on both nodes and edges
        {
            if (distance(pHeadElem->Anchors[n], TailPoint) < d)
            {
                d = distance(TailPoint, pHeadElem->Anchors[n]);
                HeadPoint = pHeadElem->Anchors[n];
            }
        }

        if (ThisFlowIsOutgoingBaggage) // i.e., this flow's is a
        baggage flow and its pHeadElem (carrier) // is another
        flow that is exiting the same function as this one
        {
            for (int n = 0; n <= 15; n++)
            {
                if (this->pTailElem->AnchorsForBaggageFlows[n] ==
                this->pHeadElem->TailPoint)
                    this->TailPoint = this->pTailElem-
                    >AnchorsForBaggageFlows[n+2];
                // The following two conditional statements are
                special cases where
                // the 14+2 = 16 and 15+2 = 17 -th elements do not
                exist in the array.
                if (this->pTailElem->AnchorsForBaggageFlows[14] ==
                this->pHeadElem->TailPoint)
                    this->TailPoint = this->pTailElem-
                    >AnchorsForBaggageFlows[0];
                if (this->pTailElem->AnchorsForBaggageFlows[15] ==
                this->pHeadElem->TailPoint)
                    this->TailPoint = this->pTailElem-
                    >AnchorsForBaggageFlows[1];
                if (this->pTailElem->AnchorsForBaggageFlows[0] ==
                this->pHeadElem->TailPoint)
                    this->TailPoint = this->pTailElem-
                    >AnchorsForBaggageFlows[2];

                this->HeadPoint = this->pHeadElem->Anchors[5];
            }
            // The following two lines improves readability of baggage
            flows
            FontSize = BAGGAGE_FONT_SIZE;
            HeadSize = EDGE_HEAD_SIZE / 2;
        }
    }

```

```

        if (ThisFlowIsIncomingBaggage)
        {
            for (int n = 0; n <= 15; n++)
            {
                if (this->pHeadElem->AnchorsForBaggageFlows[n] ==
this->pTailElem->HeadPoint)
                    this->HeadPoint = this->pHeadElem-
>AnchorsForBaggageFlows[n+2];
                // The following two conditional statements are
special cases where
                // the 14+2 = 16 and 15+2 = 17 -th elements do not
exist in the array.
                if (this->pHeadElem->AnchorsForBaggageFlows[14] ==
this->pTailElem->HeadPoint)
                    this->HeadPoint = this->pHeadElem-
>AnchorsForBaggageFlows[0];
                if (this->pHeadElem->AnchorsForBaggageFlows[15] ==
this->pTailElem->HeadPoint)
                    this->HeadPoint = this->pHeadElem-
>AnchorsForBaggageFlows[1];
                if (this->pHeadElem->AnchorsForBaggageFlows[0] ==
this->pTailElem->HeadPoint)
                    this->HeadPoint = this->pHeadElem-
>AnchorsForBaggageFlows[2];

                    this->TailPoint = this->pTailElem->Anchors[10];
            }
            // The following two lines improves readability of baggage
flows
            FontSize = BAGGAGE_FONT_SIZE;
            HeadSize = EDGE_HEAD_SIZE / 2;
        }
    }

void CEdge::DrawOnDC(CDC* pDC)
{
    AttachEdgeToNearestAnchor();

    CElement::DrawOnDC(pDC);    // Call the drawing function of the
parent class - sets pen color

    //
=====
=====
    // Draw the STEM of the arrow using PenStem (No brush required)
    //
=====
=====

    CPen PenStem;
    PenStem.CreatePen(StemLineFont, StemThickness, RGB(PenR, PenG,
PenB));

```

```

CPen* pOldPen = pDC->SelectObject(&PenStem);
CPoint* ArrowTerminalPoints = new CPoint[2];
ArrowTerminalPoints[0] = TailPoint;
ArrowTerminalPoints[1] = HeadPoint;
pDC->Polyline(ArrowTerminalPoints, 2);
delete[] ArrowTerminalPoints;

    ComputeAnchorPoints();          // Computes the eight anchor points
along the stem of                  // the edge whenever
                                   // the edge is edited, moved, or whatever.

    ResetGeometricCenter();        // Makes sure that the
GeometricCenter is reset between the // Tail and Head
points, when an arrow is moved by grabbing // Those terminal
points

    /*for (int AnchorInx = 1; AnchorInx <= 16; AnchorInx++)
        pDC->Ellipse(Ancors[AnchorInx - 1].x - 1,
Anchors[AnchorInx - 1].y - 1,
                    Anchors[AnchorInx - 1].x + 1, Anchors[AnchorInx
- 1].y + 1);*/

    //
=====
=====
    // Draw the HEAD of the arrow using PenHead and BrushHead
    //
=====
=====

CPen PenHead;
PenHead.CreatePen(PS_SOLID, THIN, RGB(PenR, PenG, PenB));

if (this->pHeadElem == NULL)
{
    HeadBrushR = DANGLING_BRUSH_R;
    HeadBrushG = DANGLING_BRUSH_G;
    HeadBrushB = DANGLING_BRUSH_B;
}
else
{
    HeadBrushR = GENERIC_BRUSH_R;
    HeadBrushG = GENERIC_BRUSH_G;
    HeadBrushB = GENERIC_BRUSH_B;
}

CBrush BrushHead(RGB(HeadBrushR, HeadBrushG, HeadBrushB));
CBrush* pOldBrush = pDC->SelectObject(&BrushHead);

pOldPen = pDC->SelectObject(&PenHead);

```

```

        double      alpha      =      atan(abs(double(HeadPoint.y) -
double(TailPoint.y)) / abs(double(HeadPoint.x) - double(TailPoint.x)));

        int X_Factor, Y_Factor;

        if (HeadPoint.x >= TailPoint.x)
            X_Factor = 1;
        else X_Factor = (-1);

        if (TailPoint.y >= HeadPoint.y)
            Y_Factor = 1;
        else Y_Factor = (-1);

        HeadLeftVertex.x = HeadPoint.x - HeadSize * cos(alpha -
HalfHeadAngle) * X_Factor;
        HeadLeftVertex.y = HeadPoint.y + HeadSize * sin(alpha -
HalfHeadAngle) * Y_Factor;

        HeadRightVertex.x = HeadPoint.x - HeadSize * cos(alpha +
HalfHeadAngle) * X_Factor;
        HeadRightVertex.y = HeadPoint.y + HeadSize * sin(alpha +
HalfHeadAngle) * Y_Factor;

        HeadVertexArray[0] = HeadPoint;
        HeadVertexArray[1] = HeadLeftVertex;
        HeadVertexArray[2] = HeadRightVertex;

        pDC->Polygon(HeadVertexArray, 3);

        //
=====
=====
        // Draw the TAIL of the arrow using PenTail and BrushTail
        //
=====
=====

        CPen PenTail;
        PenTail.CreatePen(PS_SOLID, THIN, RGB(PenR, PenG, PenB));

        if (this->pTailElem == NULL)
        {
            TailBrushR = DANGLING_BRUSH_R;
            TailBrushG = DANGLING_BRUSH_G;
            TailBrushB = DANGLING_BRUSH_B;
        }
        else
        {
            TailBrushR = GENERIC_BRUSH_R;
            TailBrushG = GENERIC_BRUSH_G;
            TailBrushB = GENERIC_BRUSH_B;
        }
    }

```

```

    CBrush BrushTail( RGB(TailBrushR, TailBrushG, TailBrushB) );
    pOldBrush = pDC->SelectObject(&BrushTail);
    pOldPen = pDC->SelectObject(&PenTail);

    pDC->Ellipse(TailPoint.x - 4, TailPoint.y - 4, TailPoint.x + 4,
TailPoint.y + 4);

    //
=====
=====
    // Put back the old objects, although I do not understand how
this impacts anything.
    //
=====
=====

    pDC->SelectObject(pOldPen);
    pDC->SelectObject(pOldBrush);
}

void CEdge::ComputeAnchorPoints()
{
    // First eight points - between tail and center
    for (int AnchorInx = 1; AnchorInx <= 16; AnchorInx++)
    {
        Anchors[AnchorInx - 1] = *InterpolatePoints(TailPoint,
HeadPoint, (0.5 / 9 * AnchorInx));
    }

    // Second eight points - between center and head
    for (int AnchorInx = 1; AnchorInx <= 8; AnchorInx++)
    {
        Anchors[AnchorInx + 7] = *InterpolatePoints(TailPoint,
HeadPoint, (0.5 + 0.5 / 9 * AnchorInx));
    }
}

void CEdge::ResetGeometricCenter()
{
    GeometricCenter = *this->InterpolatePoints(this->HeadPoint, this-
>TailPoint, 0.5);
}

```

```

#include "StdAfx.h"
#include "Element.h"

CElement::CElement(void)
{
    IsHighlighted = false;
}

```

```

    IsSelected = false;
    IsResidual = false;

    PenR = GENERIC_PEN_R;
    PenG = GENERIC_PEN_G;
    PenB = GENERIC_PEN_B;

    GrabHandle = 0; // NONE
}

CElement::~CElement(void)
{
}

void CElement::DrawOnDC(CDC* pDC)
{
    //
    =====
    // Decide the color, based on HIGHLIGHT, SELECTED, or GENERIC
    status
    //
    =====
    =====

    if (this->IsHighlighted) // This ORDER of checks is very
important. If
    { // changed, this will
change the highlight and
        this->PenR = PRESELECTION_PEN_R; // unhighlight behavior
of energies
        this->PenG = PRESELECTION_PEN_G;
        this->PenB = PRESELECTION_PEN_B;
    }

    else if (this->IsSelected)
    {
        this->PenR = SELECTION_PEN_R;
        this->PenG = SELECTION_PEN_G;
        this->PenB = SELECTION_PEN_B;
    }

    else if (this->IsResidual)
    {
        this->PenR = RESIDUAL_PEN_R;
        this->PenG = RESIDUAL_PEN_G;
        this->PenB = RESIDUAL_PEN_B;
    }

    else
    {
        this->PenR = GENERIC_PEN_R;
        this->PenG = GENERIC_PEN_G;
    }
}

```

```

        this->PenB = GENERIC_PEN_B;
    }
}

```

```

// Energize_M_Template.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Energize_M_Template.h"

// CEnergize_M_Template dialog

IMPLEMENT_DYNAMIC(CEnergize_M_Template, CDialog)

CEnergize_M_Template::CEnergize_M_Template(CWnd* pParent/* = NULL*/,
    CPoint InsertionPoint /*= (500,500)*/,
    CString* pCounterString_F /*= NULL*/,
    CString* pCounterString_InM /*= NULL*/,
    CString* pCounterString_OutM /*= NULL*/,
    CString* pCounterString_InE /*= NULL*/,
    CString* pCounterString_OutE /*= NULL*/)
    : CDialog(CEnergize_M_Template::IDD, pParent)
{
    pFunctionBlock = new CFunction(NULL, InsertionPoint,
    pCounterString_F);

    CPoint TailOfInM(InsertionPoint.x - TEMPLATE_FLOW_LENGTH,
    InsertionPoint.y);
    CPoint HeadOfOutM(InsertionPoint.x + 1.5*TEMPLATE_FLOW_LENGTH,
    InsertionPoint.y);
    CPoint TailOfInE(InsertionPoint.x, InsertionPoint.y -
    TEMPLATE_FLOW_LENGTH);

    pMaterial_InM = new CMaterial(NULL, TailOfInM, InsertionPoint,
    pCounterString_InM);
    pMaterial_OutM = new CMaterial(NULL, InsertionPoint, HeadOfOutM,
    pCounterString_OutM);
    pEnergy_InE = new CEnergy(NULL, TailOfInE, InsertionPoint,
    pCounterString_InE);
    pEnergy_OutE = new CEnergy(NULL, InsertionPoint, HeadOfOutM
    /*Dummy*/, pCounterString_OutE);

    pMaterial_InM->pHeadElem = pFunctionBlock;
    pMaterial_OutM->pTailElem = pFunctionBlock;
    pEnergy_InE->pHeadElem = pFunctionBlock;
    pEnergy_OutE->pTailElem = pFunctionBlock;
    pEnergy_OutE->pHeadElem = pMaterial_OutM;
}

```

```

CEnergize_M_Template::~CEnergize_M_Template()
{
}

void CEnergize_M_Template::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
}

BEGIN_MESSAGE_MAP(CEnergize_M_Template, CDialog)
END_MESSAGE_MAP()

// CEnergize_M_Template message handlers

```

```

// Energy.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Energy.h"

IMPLEMENT_DYNAMIC(CEnergy, CDialog)

CEnergy::CEnergy(CWnd* pParent,
                CPoint TailClick,
                CPoint HeadClick,
                CString* pCounterString,
                int ReasOpt)
    : CDialog(CEnergy::IDD, pParent)
    , GivenName(_T("E") + *pCounterString)
    , UI_IsResidual(false)
    , UI_ForceTerm(100)
    , UI_RateTerm(1)
    , ReasoningOption(ReasOpt)
{
    TailPoint = TailClick;
    HeadPoint = HeadClick;
    GeometricCenter = *InterpolatePoints(TailPoint, HeadPoint, 0.5);
    StemThickness = THIN; // This sets the thickness of Energy
arrows

    HeadSize = EDGE_HEAD_SIZE;
    HalfHeadAngle = EDGE_HEAD_HALF_ANGLE;

    ComputeAnchorPoints();
    pHeadElem = NULL;
    pTailElem = NULL;

```



```

        DoModal();          // Launches modal dialog
    }
}

BOOL CEnergy::OnInitDialog()
{
    CDialog::OnInitDialog();

    //=====
    // Grey Out Dialog Controls
    //=====

    if (this->ReasoningOption == QUALITATIVE_CONSERVATION)
        GetDlgItem(IDC_RESIDUAL_ENERGY)->EnableWindow(false);
    //Greys out control

    if ((this->ReasoningOption <  QUANTITATIVE_EFFICIENCY) /*||
(UI_IsResidual == true)*/)
    {
        GetDlgItem(IDC_FORCE_TERM)->EnableWindow(false);
        //Greys out control
        GetDlgItem(IDC_RATE_TERM)->EnableWindow(false);
        //Greys out control
        GetDlgItem(IDC_FORCE_STATIC_TEXT)->EnableWindow(false);
        //Greys out control
        GetDlgItem(IDC_RATE_STATIC_TEXT)->EnableWindow(false);
        //Greys out control
    }

    else
    {
        GetDlgItem(IDC_FORCE_TERM)->EnableWindow(true);          //
Makes control available
        GetDlgItem(IDC_RATE_TERM)->EnableWindow(true);          //
Makes control available
        GetDlgItem(IDC_FORCE_STATIC_TEXT)->EnableWindow(true);
        // Makes control available
        GetDlgItem(IDC_RATE_STATIC_TEXT)->EnableWindow(true);
        // Makes control available

        if (this->ReasoningOption >= QUANTITATIVE_EFFICIENCY)
        {
            pEnergyTaxonomy = new CTreeCtrl;

            pEnergyTaxonomy->Create(WS_CHILD      |  WS_VISIBLE      |
WS_BORDER | WS_TABSTOP |
                                TVS_HASLINES  |
TVS_HASBUTTONS | TVS_LINESATROOT |

/*TVS_SINGLEEXPAND | */TVS_SHOWSELALWAYS | TVS_TRACKSELECT,
                CRect(11, 150, 248, 440), this, 0x1221);

```

```

// Full List of all energy types (leaf nodes and
intermediate nodes)
HTREEITEM hEnergy, // Primary
    hME, hEE, hThE, hChE, hEME, hMagE, hAcE, //
Secondary
    hKinetic, hPotential, // Tertiary under hME
    hLinear, hRotatinal, // Quaternary under
hKinetic
    hGravitational, hElastic; //Quaternary under
hPotential

// PRIMARY LEVEL
hEnergy = pEnergyTaxonomy->InsertItem(_T("E"),
TVI_ROOT);

// SECONDARY LEVEL (UNDER hEnergy)
hME = pEnergyTaxonomy->InsertItem(_T("ME"), hEnergy);
hEE = pEnergyTaxonomy->InsertItem(_T("EE"), hEnergy);
hThE = pEnergyTaxonomy->InsertItem(_T("ThE"),
hEnergy);
hChE = pEnergyTaxonomy->InsertItem(_T("ChE"),
hEnergy);
hEME = pEnergyTaxonomy->InsertItem(_T("EME"),
hEnergy);
hMagE = pEnergyTaxonomy->InsertItem(_T("MagE"),
hEnergy);
hAcE = pEnergyTaxonomy->InsertItem(_T("AcE"),
hEnergy);

// TERTIARY LEVEL (UNDER hME)
hKinetic = pEnergyTaxonomy->InsertItem(_T("KE"),
hME);
hPotential = pEnergyTaxonomy->InsertItem(_T("PE"),
hME);

// QUARternary LEVEL (UNDER hME -> hKinetic)
hLinear = pEnergyTaxonomy->InsertItem(_T("LinKE"),
hKinetic);
hRotatinal = pEnergyTaxonomy->InsertItem(_T("RotKE"),
hKinetic);

// QUARternary LEVEL (UNDER hME -> hPotential)
hGravitational = pEnergyTaxonomy-
>InsertItem(_T("GrvPE"), hPotential);
hElastic = pEnergyTaxonomy->InsertItem(_T("ElPE"),
hPotential);

pEnergyTaxonomy->SelectItem(hEnergyType);

pEnergyTaxonomy->Expand(hEnergy, TVE_EXPAND);
pEnergyTaxonomy->Expand(hME, TVE_EXPAND);
pEnergyTaxonomy->Expand(hKinetic, TVE_EXPAND);
pEnergyTaxonomy->Expand(hPotential, TVE_EXPAND);

```

```

    }

    return TRUE; // return TRUE unless you set the focus to a control
}

CEnergy::~CEnergy()
{
}

void CEnergy::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_ENERGY_NAME, GivenName); // Connects
variable IDC_ENERGY_NAME to member GivenName

    DDX_Check(pDX, IDC_RESIDUAL_ENERGY, UI_IsResidual);
    DDX_Text(pDX, IDC_FORCE_TERM, UI_ForceTerm);
    DDX_Text(pDX, IDC_RATE_TERM, UI_RateTerm);
}

void CEnergy::OnOK()
{
    CDialog::OnOK();

    if (this->ReasoningOption >= QUANTITATIVE_EFFICIENCY)
    {
        hEnergyType = pEnergyTaxonomy->GetSelectedItem();
        EnergyTypeName = pEnergyTaxonomy->GetItemText(hEnergyType);
        delete pEnergyTaxonomy;
    }

    else
        EnergyTypeName = "E";
}

void CEnergy::DrawOnDC(CDC* pDC)
{
    IsResidual = UI_IsResidual;
    Power = UI_ForceTerm * UI_RateTerm;

    CEdge::DrawOnDC(pDC); // Execute the entire drawing code
of the parent class CEdge

    //
=====
=====
    // Write the name of CEnergy using a Font object
    //
=====
=====

    // Initializes a CFont object with the specified characteristics.

```

```

CFont font;
VERIFY(font.CreateFont(
    FontSize,                                // nHeight
    0,                                        // nWidth
    0,                                        // nEscapement
    0,                                        // nOrientation
    FW_NORMAL,                                // nWeight
    FALSE,                                    // bItalic
    FALSE,                                    // bUnderline
    0,                                        // cStrikeOut
    ANSI_CHARSET,                            // nCharSet
    OUT_DEFAULT_PRECIS,                      // nOutPrecision
    CLIP_DEFAULT_PRECIS,                    // nClipPrecision
    DEFAULT_QUALITY,                         // nQuality
    DEFAULT_PITCH | FF_SWISS,               // nPitchAndFamily
    _T("Arial")));                          // lpszFacename

CFont* def_font = pDC->SelectObject(&font);
pDC->SetTextAlign(TA_CENTER | TA_BASELINE);

CString* pPowerString = new CString;
*pPowerString = "";

if ((this->ReasoningOption >= QUANTITATIVE_EFFICIENCY) /*&&
(this->IsResidual == false)*/)
{
    pPowerString->Format(_T("%2.0f"), this->Power);
    *pPowerString = *pPowerString + _T("W");
}
else
    *pPowerString = "";

pDC->TextOut(GeometricCenter.x, GeometricCenter.y, (GivenName + "
[" + EnergyTypeName + "] " + *pPowerString));
pDC->SelectObject(def_font);

//
=====
// Put back the old objects, although I do not understand how
this impacts anything.
//
=====
font.DeleteObject();
delete pPowerString;
}

BEGIN_MESSAGE_MAP(CEnergy, CDialog)
END_MESSAGE_MAP()

```

```

#include "stdafx.h"
#include "ConMod.h"
#include "Env.h"
#include "geometry.h"

// CEnv dialog

IMPLEMENT_DYNAMIC(CEnv, CDialog)

CEnv::CEnv(CWnd* pParent, CPoint InsertionPoint, CString*
pCounterString)
    : CDialog(CEnv::IDD, pParent)
    , GivenName("Env" + *pCounterString)
{
    GeometricCenter = InsertionPoint;
    ComputeBlockCoordinates();

    DoModal(); // Launches modal dialog
}

CEnv::~CEnv()
{
}

void CEnv::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_ENV_NAME, GivenName);
}

void CEnv::ComputeBlockCoordinates()
{
    Anchors[0] = CPoint ((GeometricCenter.x + ENV_SIZE),
(GeometricCenter.y));
    Anchors[1] = CPoint ((GeometricCenter.x), (GeometricCenter.y +
ENV_SIZE * 0.866));
    Anchors[2] = CPoint ((GeometricCenter.x - ENV_SIZE),
(GeometricCenter.y));
    Anchors[3] = CPoint ((GeometricCenter.x), (GeometricCenter.y -
ENV_SIZE * 0.866));
    Anchors[4] = CPoint ((GeometricCenter.x + 0.5*ENV_SIZE),
(GeometricCenter.y + ENV_SIZE * 0.866));
    Anchors[5] = CPoint ((GeometricCenter.x - 0.5*ENV_SIZE),
(GeometricCenter.y + ENV_SIZE * 0.866));
    Anchors[6] = CPoint ((GeometricCenter.x - 0.5*ENV_SIZE),
(GeometricCenter.y - ENV_SIZE * 0.866));
    Anchors[7] = CPoint ((GeometricCenter.x + 0.5*ENV_SIZE),
(GeometricCenter.y - ENV_SIZE * 0.866));
}

```

```

    Anchors[8] = CPoint ((GeometricCenter.x + 0.75*ENV_SIZE),
(GeometricCenter.y + ENV_SIZE * 0.433));
    Anchors[9] = CPoint ((GeometricCenter.x - 0.75*ENV_SIZE),
(GeometricCenter.y + ENV_SIZE * 0.433));
    Anchors[10] = CPoint ((GeometricCenter.x - 0.75*ENV_SIZE),
(GeometricCenter.y - ENV_SIZE * 0.433));
    Anchors[11] = CPoint ((GeometricCenter.x + 0.75*ENV_SIZE),
(GeometricCenter.y - ENV_SIZE * 0.433));
    Anchors[12] = Anchors[0];
    Anchors[13] = Anchors[0];
    Anchors[14] = Anchors[0];
    Anchors[15] = Anchors[0];

    AnchorsForBaggageFlows[0] = Anchors[0];
    AnchorsForBaggageFlows[1] = Anchors[11];
    AnchorsForBaggageFlows[2] = Anchors[7];
    AnchorsForBaggageFlows[3] = Anchors[3];
    AnchorsForBaggageFlows[4] = Anchors[6];
    AnchorsForBaggageFlows[5] = Anchors[10];
    AnchorsForBaggageFlows[6] = Anchors[2];
    AnchorsForBaggageFlows[7] = Anchors[9];
    AnchorsForBaggageFlows[8] = Anchors[5];
    AnchorsForBaggageFlows[9] = Anchors[1];
    AnchorsForBaggageFlows[10] = Anchors[4];
    AnchorsForBaggageFlows[11] = Anchors[8];
    AnchorsForBaggageFlows[12] = Anchors[0];
    AnchorsForBaggageFlows[13] = Anchors[0];
    AnchorsForBaggageFlows[14] = Anchors[0];
    AnchorsForBaggageFlows[15] = Anchors[0];
}

void CEnv::DrawOnDC(CDC* pDC)
{
    CElement::DrawOnDC(pDC); // Call the drawing function of the
parent class - sets pen color

    if (this->NoInputAttached && this->NoOutputAttached)
    {
        BrushR = DANGLING_BRUSH_R;
        BrushG = DANGLING_BRUSH_G;
        BrushB = DANGLING_BRUSH_B;
    }
    else
    {
        BrushR = ENV_BRUSH_R;
        BrushG = ENV_BRUSH_G;
        BrushB = ENV_BRUSH_B;
    }

    CPen Pen;
    Pen.CreatePen(PS_SOLID, 2, RGB(PenR, PenG, PenB));
    CPen* pOldPen = pDC->SelectObject(&Pen);
    CBrush Brush(RGB(BrushR, BrushG, BrushB));

```

```

CBrush* pOldBrush = pDC->SelectObject(&Brush);

ComputeBlockCoordinates();

CPoint AnchorsForHexagon[6];
AnchorsForHexagon[0] = Anchors[0];
AnchorsForHexagon[1] = Anchors[4];
AnchorsForHexagon[2] = Anchors[5];
AnchorsForHexagon[3] = Anchors[2];
AnchorsForHexagon[4] = Anchors[6];
AnchorsForHexagon[5] = Anchors[7];
pDC->Polygon(AnchorsForHexagon, 6);

//pDC->Ellipse(GeometricCenter.x - 2, GeometricCenter.y - 2,
//GeometricCenter.x + 2, GeometricCenter.y + 2);
/*
for (int AnchorInx = 1; AnchorInx <= 16; AnchorInx++)
    pDC->Ellipse(Anchors[AnchorInx - 1].x - 1,
Anchors[AnchorInx - 1].y - 1,
                Anchors[AnchorInx - 1].x + 1, Anchors[AnchorInx
- 1].y + 1);*/

// Initializes a CFont object with the specified characteristics.
CFont font;
VERIFY(font.CreateFont(
    16,                // nHeight
    0,                 // nWidth
    0,                 // nEscapement
    0,                 // nOrientation
    FW_NORMAL,        // nWeight
    FALSE,             // bItalic
    FALSE,             // bUnderline
    0,                 // cStrikeOut
    ANSI_CHARSET,     // nCharSet
    OUT_DEFAULT_PRECIS, // nOutPrecision
    CLIP_DEFAULT_PRECIS, // nClipPrecision
    DEFAULT_QUALITY,  // nQuality
    DEFAULT_PITCH | FF_SWISS, // nPitchAndFamily
    _T("Arial")));    // lpszFacename

CFont* def_font = pDC->SelectObject(&font);
pDC->SetTextAlign(TA_CENTER | TA_BASELINE);
pDC->TextOut(GeometricCenter.x, GeometricCenter.y, GivenName);
pDC->SelectObject(def_font);

font.DeleteObject();
pDC->SelectObject(pOldPen);
pDC->SelectObject(pOldBrush);
}

BEGIN_MESSAGE_MAP(CEnv, CDialog)
END_MESSAGE_MAP()

```

```
// CEnv message handlers
```

```
#include "stdafx.h"
#include "ConMod.h"
#include "Function.h"

// CFunction dialog

IMPLEMENT_DYNAMIC(CFunction, CDialog)

CFunction::CFunction(CWnd* pParent, CPoint InsertionPoint, CString*
pCounterString)
    : CDialog(CFunction::IDD, pParent)
    , GivenName("F" + *pCounterString) // Populates the string
in the GivenName field - default
{
    GeometricCenter = InsertionPoint;
    ComputeBlockCoordinates();
    Efficiency = 0;

    DoModal(); // Launches modal dialog
}

CFunction::~CFunction()
{
}

void CFunction::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_FUNCTION_NAME, GivenName); // Connects variable
with dialog element
}

BEGIN_MESSAGE_MAP(CFunction, CDialog)

END_MESSAGE_MAP()

void CFunction::ComputeBlockCoordinates()
{
    left = GeometricCenter.x - BLOCK_LENGTH / 2;
    right = GeometricCenter.x + BLOCK_LENGTH / 2;
    top = GeometricCenter.y - BLOCK_HEIGHT / 2;
    bottom = GeometricCenter.y + BLOCK_HEIGHT / 2;

    Anchors[0] = CPoint (right, (top + bottom)/2); // E
    Anchors[4] = CPoint (right, top); // NE
    Anchors[1] = CPoint ((right + left)/2, top); // N
    Anchors[5] = CPoint (left, top); // NW
}
```



```

    Anchors[2] = CPoint (left, (top + bottom)/2);    // W
    Anchors[6] = CPoint (left, bottom);              // SW
    Anchors[3] = CPoint ((left + right)/2, bottom); // S
    Anchors[7] = CPoint (right, bottom);            // SE

    Anchors[8] = CPoint (right, top + BLOCK_HEIGHT/4); // ENE
    Anchors[9] = CPoint (right - BLOCK_LENGTH/4, top); // NNE
    Anchors[10] = CPoint (left + BLOCK_LENGTH/4, top); //
NNW
    Anchors[11] = CPoint (left, top + BLOCK_HEIGHT/4); //
WNW

    Anchors[12] = CPoint (left, bottom - BLOCK_HEIGHT/4); // WSW
    Anchors[13] = CPoint (left + BLOCK_LENGTH/4, bottom); // SSW
    Anchors[14] = CPoint (right - BLOCK_LENGTH/4, bottom); // SSE
    Anchors[15] = CPoint (right, bottom - BLOCK_HEIGHT/4); //
ESE

    // The following lines reorders the anchors to a different list,
    // AnchorsForBaggageFlows. This list is scrolled through when a
    // baggage flow (incoming or outgoing) needs to be attached to
    // the function with only two nodes apart from where the main
    // flow is attached.

    AnchorsForBaggageFlows[0] = Anchors[0];
    AnchorsForBaggageFlows[1] = Anchors[8];
    AnchorsForBaggageFlows[2] = Anchors[4];
    AnchorsForBaggageFlows[3] = Anchors[9];
    AnchorsForBaggageFlows[4] = Anchors[1];
    AnchorsForBaggageFlows[5] = Anchors[10];
    AnchorsForBaggageFlows[6] = Anchors[5];
    AnchorsForBaggageFlows[7] = Anchors[11];
    AnchorsForBaggageFlows[8] = Anchors[2];
    AnchorsForBaggageFlows[9] = Anchors[12];
    AnchorsForBaggageFlows[10] = Anchors[6];
    AnchorsForBaggageFlows[11] = Anchors[13];
    AnchorsForBaggageFlows[12] = Anchors[3];
    AnchorsForBaggageFlows[13] = Anchors[14];
    AnchorsForBaggageFlows[14] = Anchors[7];
    AnchorsForBaggageFlows[15] = Anchors[15];
}

void CFunction::DrawOnDC(CDC* pDC)
{
    CElement::DrawOnDC(pDC); // Call the drawing function of the
parent class - sets pen color

    if (this->NoInputAttached && this->NoOutputAttached)
    {
        BrushR = DANGLING_BRUSH_R;
        BrushG = DANGLING_BRUSH_G;
        BrushB = DANGLING_BRUSH_B;
    }
}

```

```

}
else
{
    BrushR = FUNCTION_BRUSH_R;
    BrushG = FUNCTION_BRUSH_G;
    BrushB = FUNCTION_BRUSH_B;
}

CPen Pen;
Pen.CreatePen(PS_SOLID, 2, RGB(PenR, PenG, PenB));
CPen* pOldPen = pDC->SelectObject(&Pen);
CBrush Brush(RGB(BrushR, BrushG, BrushB));
CBrush* pOldBrush = pDC->SelectObject(&Brush);

ComputeBlockCoordinates();
CRect VerbRect(left, top, right, bottom);
pDC->Rectangle(VerbRect);

//pDC->Ellipse(GeometricCenter.x - 2, GeometricCenter.y - 2,
//GeometricCenter.x + 2, GeometricCenter.y + 2);
/*
for (int AnchorInx = 1; AnchorInx <= 16; AnchorInx++)
    pDC->Ellipse(Anchors[AnchorInx - 1].x - 1,
Anchors[AnchorInx - 1].y - 1,
                Anchors[AnchorInx - 1].x + 1, Anchors[AnchorInx
- 1].y + 1);*/

// Initializes a CFont object with the specified characteristics.
CFont font;
VERIFY(font.CreateFont(
    16,                // nHeight
    0,                 // nWidth
    0,                 // nEscapement
    0,                 // nOrientation
    FW_NORMAL,        // nWeight
    FALSE,             // bItalic
    FALSE,             // bUnderline
    0,                 // cStrikeOut
    ANSI_CHARSET,     // nCharSet
    OUT_DEFAULT_PRECIS, // nOutPrecision
    CLIP_DEFAULT_PRECIS, // nClipPrecision
    DEFAULT_QUALITY,  // nQuality
    DEFAULT_PITCH | FF_SWISS, // nPitchAndFamily
    _T("Arial")));    // lpszFacename

CFont* def_font = pDC->SelectObject(&font);
pDC->SetTextAlign(TA_CENTER | TA_BASELINE);
pDC->TextOut(GeometricCenter.x, GeometricCenter.y, GivenName);
pDC->SelectObject(def_font);

font.DeleteObject();
pDC->SelectObject(pOldPen);
pDC->SelectObject(pOldBrush);

```

```
}
```

```
#include "StdAfx.h"
#include "Geometry.h"

#define GRID_SIZE 20

CGeometry::CGeometry(void)
{
}

CGeometry::~CGeometry(void)
{
}

int CGeometry::RoundToInteger(long Coordinate, int GridSize)
{
    int GridCountLower = int(Coordinate)/GridSize;
    if ((Coordinate - GridCountLower * GridSize) <= (GridSize / 2))
        return (GridCountLower * GridSize);
    else return (GridCountLower * GridSize + GridSize);
}

CPoint CGeometry::SnapToGrid(CPoint p)
{
    return CPoint(RoundToInteger(p.x, GRID_SIZE), RoundToInteger(p.y,
GRID_SIZE));
}

long CGeometry::distance(CPoint p1, CPoint p2)
{
    return sqrt(pow((p1.x - p2.x), 2.0) + pow((p1.y - p2.y), 2.0));
}

CPoint* CGeometry::InterpolatePoints(CPoint p1, CPoint p2, double
ratio)
{
    long x_new = ((p2.x - p1.x) * ratio) + p1.x;
    long y_new = ((p2.y - p1.y) * ratio) + p1.y;
    CPoint NewPoint(x_new, y_new);
    return &NewPoint;
}/**/
```

```
// MainFrm.cpp : implementation of the CMainFrame class
//
#include "stdafx.h"
```

```

#include "ConMod.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    ON_WM_CREATE()
    // Global help commands
    ON_COMMAND(ID_HELP_FINDER, &CMDIFrameWnd::OnHelpFinder)
    ON_COMMAND(ID_HELP, &CMDIFrameWnd::OnHelp)
    ON_COMMAND(ID_CONTEXT_HELP, &CMDIFrameWnd::OnContextHelp)
    ON_COMMAND(ID_DEFAULT_HELP, &CMDIFrameWnd::OnHelpFinder)
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here
}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {

```

```

        TRACE0("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
            sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;        // fail to create
    }

    // TODO: Delete these three lines if you don't want the toolbar
to be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    // Custom ConMod toolbar controls:  PRIMITIVES TOOLBAR

    if (!m_primitivesToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_LEFT
        | CBRS_GRIPPER    | CBRS_TOOLTIPS    | CBRS_FLYBY    |
CBRS_SIZE_DYNAMIC) ||
        !m_primitivesToolBar.LoadToolBar(IDR_PRIMITIVES))
    {
        TRACE0("Failed to create PRIMITIVES toolbar\n");
        return -1;        // fail to create
    }

    m_primitivesToolBar.EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_primitivesToolBar);

    // Custom ConMod toolbar controls:  FEATURES TOOLBAR

    //=====
    // Commented out for rolling back to Layer 1 (Chapter 6)
    //=====

    if (!m_featuresToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_BOTTOM
        | CBRS_GRIPPER    | CBRS_TOOLTIPS    | CBRS_FLYBY    |
CBRS_SIZE_DYNAMIC) ||
        !m_featuresToolBar.LoadToolBar(IDR_FEATURES))
    {
        TRACE0("Failed to create FEATURES toolbar\n");
        return -1;        // fail to create
    }

    m_featuresToolBar.EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_featuresToolBar);

    // Custom ConMod toolbar controls:  REASONING TOOLBAR

```

```

        if (!m_reasoningToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_RIGHT
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) ||
        !m_reasoningToolBar.LoadToolBar(IDR_REASONING))
        {
            TRACE0("Failed to create REASONING toolbar\n");
            return -1; // fail to create
        }

        m_reasoningToolBar.EnableDocking(CBRS_ALIGN_ANY);
        DockControlBar(&m_reasoningToolBar);

        return 0;
    }

    BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
    {
        if( !CMDIFrameWnd::PreCreateWindow(cs) )
            return FALSE;
        // TODO: Modify the Window class or styles here by modifying
        // the CREATESTRUCT cs

        // Control the size of the main frame window
        BOOL bRet = CFrameWnd::PreCreateWindow(cs);
        cs.cx = 1200;
        cs.cy = 800;
        return bRet;
        //return TRUE;
    }

    // CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CMDIFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CMDIFrameWnd::Dump(dc);
}

#endif // _DEBUG

// CMainFrame message handlers

```

```

// Material.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Material.h"

IMPLEMENT_DYNAMIC(CMaterial, CDialog)

CMaterial::CMaterial(CWnd* pParent,
                    CPoint TailClick,
                    CPoint HeadClick,
                    CString* pCounterString,
                    int ReasOpt)
    : CDialog(CMaterial::IDD, pParent)
    , GivenName(_T("M") + *pCounterString)
    , UI_IsResidual(false)
    , ReasoningOption(ReasOpt)
{
    TailPoint = TailClick;
    HeadPoint = HeadClick;
    GeometricCenter = *InterpolatePoints(TailPoint, HeadPoint, 0.5);
    StemThickness = MEDIUM; // This sets the thickness of Material
arrows

    HeadSize = EDGE_HEAD_SIZE;
    HalfHeadAngle = EDGE_HEAD_HALF_ANGLE;

    ComputeAnchorPoints();
    pHeadElem = NULL;
    pTailElem = NULL;

    DoModal(); // Launches modal dialog
}

BOOL CMaterial::OnInitDialog()
{
    CDialog::OnInitDialog();

    //=====
    // Grey Out Dialog Controls
    //=====

    if (this->ReasoningOption == QUALITATIVE_CONSERVATION)
        GetDlgItem(IDC_RESIDUAL_MATERIAL)->EnableWindow(false);
    //Greys out control

    if (this->ReasoningOption >= QUANTITATIVE_EFFICIENCY)
    {
        pMaterialTaxonomy = new CTreeCtrl;

        pMaterialTaxonomy->Create(WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP |

```

```

TVS_HASBUTTONS | TVS_LINESATROOT |
TVS_HASLINES |
/*TVS_SINGLEEXPAND | */TVS_SHOWSELALWAYS | TVS_TRACKSELECT,
    CRect(11, 60, 248, 150), this, 0x1221);

    // Full List of all energy types (leaf nodes and
intermediate nodes)
    HTREEITEM hMaterial, // Primary
        hSolid, hLiquid, hGaseous; //Secondary under
hMaterial

    // PRIMARY LEVEL
    hMaterial = pMaterialTaxonomy->InsertItem(_T("M"),
TVI_ROOT);

    // SECONDARY LEVEL (UNDER hMaterial)
    hSolid = pMaterialTaxonomy->InsertItem(_T("S"), hMaterial);
    hLiquid = pMaterialTaxonomy->InsertItem(_T("L"),
hMaterial);
    hGaseous = pMaterialTaxonomy->InsertItem(_T("G"),
hMaterial);

    pMaterialTaxonomy->SelectItem(hMaterialType);
    pMaterialTaxonomy->Expand(hMaterial, TVE_EXPAND);
}

    return TRUE; // return TRUE unless you set the focus to a control
}

CMaterial::~CMaterial()
{
    //delete pMaterialTaxonomy;
}

void CMaterial::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_MATERIAL_NAME, GivenName); // Connects
variable IDC_MATERIAL_NAME to member GivenName

    DDX_Check(pDX, IDC_RESIDUAL_MATERIAL, UI_IsResidual);
}

void CMaterial::OnOK()
{
    CDialog::OnOK();

    if (this->ReasoningOption >= QUANTITATIVE_EFFICIENCY)
    {
        hMaterialType = pMaterialTaxonomy->GetSelectedItem();
        MaterialTypeName = pMaterialTaxonomy-
>GetItemText(hMaterialType);
    }
}

```



```

        delete pMaterialTaxonomy;
    }

    else
        MaterialTypeName = "M";
}

void CMaterial::DrawOnDC(CDC* pDC)
{
    IsResidual = UI_IsResidual;

    CEdge::DrawOnDC(pDC);          // Execute the entire drawing code
of the parent class CEdge

    //
=====
=====
    // Write the name of CMaterial using a Font object
    //
=====
=====

    // Initializes a CFont object with the specified characteristics.
    CFont font;
    VERIFY(font.CreateFont(
        FontSize,                      // nHeight
        0,                              // nWidth
        0,                              // nEscapement
        0,                              // nOrientation
        FW_NORMAL,                      // nWeight
        FALSE,                          // bItalic
        FALSE,                          // bUnderline
        0,                              // cStrikeOut
        ANSI_CHARSET,                  // nCharSet
        OUT_DEFAULT_PRECIS,            // nOutPrecision
        CLIP_DEFAULT_PRECIS,          // nClipPrecision
        DEFAULT_QUALITY,              // nQuality
        DEFAULT_PITCH | FF_SWISS,     // nPitchAndFamily
        _T("Arial")));                // lpszFacename

    CFont* def_font = pDC->SelectObject(&font);
    pDC->SetTextAlign(TA_CENTER | TA_BASELINE);
    pDC->TextOut(GeometricCenter.x, GeometricCenter.y, (GivenName + "
[" + MaterialTypeName + "]"));
    pDC->SelectObject(def_font);

    //
=====
=====
    // Put back the old objects, although I do not understand how
this impacts anything.

```

```

//
=====
font.DeleteObject();
}
BEGIN_MESSAGE_MAP(CMaterial, CDialog)
END_MESSAGE_MAP()

```

```

#include "StdAfx.h"
#include "Node.h"

CNode::CNode(void)
{
}

CNode::~~CNode(void)
{
}

void CNode::ComputeBlockCoordinates()
{}

/*void CNode::DrawOnDC(CDC* pDC)
{*/

```

```

// Signal.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Signal.h"

// CSignal dialog

IMPLEMENT_DYNAMIC(CSignal, CDialog)

CSignal::CSignal(CWnd* pParent, CPoint TailClick, CPoint HeadClick,
CString* pCounterString)
: CDialog(CSignal::IDD, pParent)
, GivenName(_T("S") + *pCounterString)

```

```

{
    TailPoint = TailClick;
    HeadPoint = HeadClick;
    GeometricCenter = *InterpolatePoints(TailPoint, HeadPoint, 0.5);
    StemThickness = THIN;    // This sets the thickness of signal
arrows
    StemLineFont = PS_DOT;

    HeadSize = EDGE_HEAD_SIZE;
    HalfHeadAngle = EDGE_HEAD_HALF_ANGLE;

    ComputeAnchorPoints();
    pHeadElem = NULL;
    pTailElem = NULL;

    DoModal();    // Launches modal dialog
}

CSignal::~CSignal()
{
}

void CSignal::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    DDX_Text(pDX, IDC_SIGNAL_NAME, GivenName);    // Connects
variable IDC_SIGNAL_NAME to member GivenName
}

void CSignal::DrawOnDC(CDC* pDC)
{
    CEdge::DrawOnDC(pDC);    // Execute the entire drawing code
of the parent class CEdge

    //
=====
=====
    // Write the name of CMaterial using a Font object
    //
=====
=====

    // Initializes a CFont object with the specified characteristics.
    CFont font;
    VERIFY(font.CreateFont(
        FontSize,    // nHeight
        0,    // nWidth
        0,    // nEscapement
        0,    // nOrientation
        FW_NORMAL,    // nWeight
        FALSE,    // bItalic
        FALSE,    // bUnderline
        0,    // cStrikeOut

```

```

        ANSI_CHARSET,           // nCharSet
        OUT_DEFAULT_PRECIS,     // nOutPrecision
        CLIP_DEFAULT_PRECIS,    // nClipPrecision
        DEFAULT_QUALITY,        // nQuality
        DEFAULT_PITCH | FF_SWISS, // nPitchAndFamily
        _T("Arial"));          // lpszFacename

    CFont* def_font = pDC->SelectObject(&font);
    pDC->SetTextAlign(TA_CENTER | TA_BASELINE);
    pDC->TextOut(GeometricCenter.x, GeometricCenter.y, GivenName);
    pDC->SelectObject(def_font);

    //
    =====
    =====
    // Put back the old objects, although I do not understand how
    this impacts anything.
    //
    =====
    =====

    font.DeleteObject();
}

BEGIN_MESSAGE_MAP(CSignal, CDialog)
END_MESSAGE_MAP()

// CSignal message handlers

```

```

// stdafx.cpp : source file that includes just the standard includes
// ConMod.pch will be the pre-compiled header
// stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"

```

```

// Template.cpp : implementation file
//

#include "stdafx.h"
#include "ConMod.h"
#include "Template.h"

// CTemplate

```

```
CTemplate::CTemplate()  
{  
}  
  
CTemplate::~~CTemplate()  
{  
}
```

## REFERENCES

- [1] Pahl, G., Beitz, W., Feldhusen, J., and Grote, K. H., 2007, *Engineering Design: A Systematic Approach*, 3rd. ed. Springer-Verlag London Limited. London, UK.
- [2] Otto, K. N. and Wood, K. L., 2001, *Product Design Techniques in Reverse Engineering and New Product Development* Prentice Hall. Upper Saddle River, NJ.
- [3] Ullman, D. G., 1992, *The Mechanical Design Process* McGraw-Hill. New York.
- [4] Ulrich, K. T. and Eppinger, S. D., 2008, *Product Design and Development*, Fourth ed. McGraw-Hill. New York, NY, USA.
- [5] Sridharan, P. and Campbell, M. I., 2004, "A Grammar for Function Structures," *ASME 2004 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Salt Lake City, UT, USA, September 28-October 2, 2004.
- [6] Sridharan, P. and Campbell, M. I., 2005, "A Study on the Grammatical Construction of Function Structures," *Artificial Intelligence for Engineering Design, Analysis & Manufacturing*, **19**(3) pp. 139-160.
- [7] Chandrasekaran, B., 1990, "Design Problem Solving: A Task Analysis," *AI Magazine*, **11**(4) pp. 59-71.
- [8] Gero, J. S., 1990, "Design Prototypes: A Knowledge Representation Schema for Design," *AI Magazine*, **11**(4) pp. 26-36.
- [9] Gero, J. S., 1996, "Creativity, Emergence and Evolution in Design," *Knowledge-Based Systems*, **9**(7) pp. 435-448.
- [10] Gero, J. S. and Kannengiesser, U., 2000, "Towards a Situated Function-Behaviour-Structure Framework as the Basis for a Theory of Designing,"

- Workshop on Development and Application of Design Theories in AI in Design Research, Sixth International Conference on Artificial Intelligence in Design, Worcester, MA, USA, June 2000.*
- [11] Gero, J. S. and Kannengiesser, U., "The Situated Function-Behaviour-Structure Framework," in *Artificial Intelligence in Design*, J. S. Gero, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2002, pp. 89-104.
- [12] Goel, A. K. and Bhatta, S. R., 2004, "Use of Design Patterns in Analogy-Based Design," *Advanced Engineering Informatics*, **18**pp. 85-94.
- [13] Iwasaki, Y., Fikes, R., Vescovi, M., and Chandrasekaran, B., 1993, "How Things Are Intended to Work: Capturing Functional Knowledge in Device Design," *International Joint Conference on Artificial Intelligence*, Menlo Park, CA.
- [14] Schultz, R. R., Nigg, D. W., Ougouag, A. M., Terry, W. K., Wolf, J. R., Gougar, H. D., Johnsen, G. W., McEligot, D. M., McCreery, G. E., Johnson, R. W., Sterbentz, J. W., and MacDonald, P. E., "Next Generation Nuclear Plant—Design Methods Development and Validation Research and Development Program Plan," Idaho National Engineering and Environmental Laboratory, Idaho Falls, ID September 2004 2004.
- [15] Goldschmidt, G., 1991, "The Dialectics of Sketching," *Creativity Research Journal*, **4**(2) pp. 123-143.
- [16] Umeda, Y., Ishii, M., Yoshioka, M., Shimomura, Y., and Tomiyama, T., 1996, "Supporting Conceptual Design Based on the Function-Behavior-State Modeler," *Artificial Intelligence for Engineering Design, Analysis & Manufacturing*, **10**(4) pp. 275-288.
- [17] Umeda, Y. and Tomiyama, T., 1995, "FBS Modeling: Modeling Scheme of Function for Conceptual Design," *9th. International Workshop on Qualitative Reasoning*, Amsterdam, Netherlands, May 1995.
- [18] Umeda, Y. and Tomiyama, T., 1997, "Functional Reasoning in Design," *IEEE Intelligent Systems*, **12**(2) pp. 42-48.

- [19] Vescovi, M., Iwasaki, Y., Fikes, R., and Chandrasekaran, B., 1993, "CFRL: A Language for Specifying the Causal Functionality of Engineered Devices," *Eleventh National Conference on Artificial Intelligence*, Washington, D.C., July 1993.
- [20] Kurtoglu, T., "A Computational Approach to Innovative Conceptual Design," in *Mechanical Engineering*. vol. PhD Austin: University of Texas, 2007, p. 155.
- [21] Kurtoglu, T., Campbell, M. I., Gonzales, J., Bryant, C. R., and Stone, R. B., 2005, "Capturing Empirically Derived Design Knowledge for Creating Conceptual Design Configurations," *ASME 2005 International Design Engineering and Technical Conferences and Computers and Information in Engineering Conference*, Long Beach, CA, USA, September 24-28, 2005.
- [22] Kurtoglu, T., Swantner, A., and Campbell, M. I., 2010, "Automating the Conceptual Design Process: From Black Box to Component Selection," *Artificial Intelligence for Engineering Design, Analysis & Manufacturing*, **24**(1) pp. 49-62.
- [23] Ullman, D. G., Dietterich, T. G., and Stauffer, L. A., 1988, "A Model of the Mechanical Design Process Based on Empirical Data," *Artificial intelligence for Engineering Design, Analysis, and Manufacturing*, **2**(1) pp. 33-52.
- [24] Ullman, D. G., Wood, S., and Craig, D., 1990, "The Importance of Drawing in the Mechanical Design Process," *Computer & Graphics*, **14**(2) pp. 163-274.
- [25] Stone, R. B. and Wood, K. L., 2000, "Development of a Functional Basis for Design," *Journal of Mechanical Design*, **122**(4) pp. 359-370.
- [26] Hirtz, J., Stone, R. B., McAdams, D. A., Szykman, S., and Wood, K. L., 2002, "A Functional Basis for Engineering Design: Reconciling and Evolving Previous Efforts," *Research in Engineering Design*, **13**(2) pp. 65-82.
- [27] Kitamura, Y., Kashiwaseb, M., Fuseb, M., and Mizoguchi, R., 2004, "Deployment of an Ontological Framework of Functional Design Knowledge," *Advanced Engineering Informatics*, **18**(2) pp. 115-127.



- [28] Kitamura, Y., Koji, Y., and Mizoguchi, R., 2005, "An Ontological Model of Device Function and Its Deployment for Engineering Knowledge Sharing," *First Workshop FOMI 2005 - Formal Ontologies Meet Industry*, Castelnuovo del Garda (VR), Italy, June 9-10, 2005.
- [29] Kitamura, Y. and Mizoguchi, R., 2003, "Ontology-Based Description of Functional Design Knowledge and Its Use in a Functional Way Server," *Expert Systems with Applications*, **24**(2003) pp. 153-166.
- [30] Summers, J. D., 2005, "Reasoning in Engineering Design," *ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Long Beach, California.
- [31] Summers, J. D. and Shah, J. J., 2004, "Representation in Engineering Design: A Framework for Classification," *ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Salt Lake City, UT, Sep 28 - Oct 2, 2004.
- [32] Baumgart, B. G., "Winged Edge Polyhedron Representation," Stanford University, Stanford, CA, USA 1972.
- [33] Baumgart, B. G., "Geometric Modelling for Computer Vision," Stanford University Stanford, CA, USA Artificial Intelligence Report Number CS-463, 1974.
- [34] Corney, J. and Lim, T., 2002, *3d Modeling with Acis*, Second ed. Saxe-Coburg Publications.
- [35] Dym, C. L., 1994, "Representing Designed Artifacts: The Languages of Engineering Design," *Archives of Computational Methods in Engineering*, **1**pp. 75-108.
- [36] Dym, C., 1995, *Engineering Design: A Synthesis of Views* Cambridge University Press. New York, NY.

- [37] Dym, C. L., 1992, "Representation and Problem-Solving: The Foundations of Engineering Design," *Environment and Planning B: Planning and Design*, **19**(1) pp. 97-105.
- [38] Dym, C. L., Agogino, A. M., Eris, O., Frey, D. D., and Leifer, L. J., 2005, "Engineering Design Thinking, Teaching, and Learning," *Journal of Engineering Education*, **94**(1) pp. 103-120.
- [39] Regli, W., Hu, X., Atwood, M., and Sun, W., 2000, "A Survey of Design Rationale Systems: Approaches, Representation, Capture, and Retrieval," *Engineering with Computers*, **16**pp. 209-235.
- [40] Ullman, D. and Abmrosio, B., 1995, "A Taxonomy for Classifying Engineering Decision Problems and Support Systems," *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, **9**(427-438)
- [41] Shah, J. and Wilson, P., 1989, "Analysis of Design Abstraction, Representation, and Inferencing Requirements for Computer Aided Design," *Journal of Design Studies*, **10**(3) pp. 169-178.
- [42] Finger, S. and Dixon, J. R., 1989, "A Review of Research in Mechanical Engineering Design. Part II: Representations, Analysis, and Design for the Life Cycle," *Research in engineering design*, **1**(121-137)
- [43] Pratt, M. J., 2001, "Introduction to Iso 10303 - the Step Standard for Product Data Exchange," *Journal of Computing and Information Science in Engineering*, **1**(1) pp. 102-103.
- [44] Summers, J. D., "Expressiveness of the Design Exemplar," in *ASME 2005 International Design Engineering & Technical Conferences & Computers and Information in Engineering Conferences*. vol. CIE-85135 Long Beach, California, USA: ASME, 2005.
- [45] Summers, J. D., Bettig, B., and Shah, J. J., 2004, "The Design Exemplar: A New Data Structure for Embodiment Design Automation," *ASME Journal of Mechanical Design*, **126**(5) pp. 775-787.

- [46] Minsky, M. L., "Matter, Mind and Models," in *IFIP Congress*. vol. 1 Washington, D.C., USA, 1965, pp. 45-49.
- [47] Szykman, S., Racz, J. W., and Sriram, R. D., 1999, "The Representation of Function in Computer-Based Design," *1999 ASME Design Engineering Technical Conferences*, Las Vegas, NV, USA, September 12-15, 1999.
- [48] Sen, C., Summers, J. D., and Mocko, G. M., 2011, "A Protocol to Formalize Function Verbs to Support Conservation-Based Model Check Reasoning," *Journal of Engineering Design (In Press)*,
- [49] Diestel, R., 2005, *Graph Theory*, 3rd. ed. Springer-Verlag. Heidelberg, New York.
- [50] Bohm, M. R. and Stone, R. B., 2004, "Product Design Support: Exploring a Design Repository System," *ASME International Mechanical Engineering Congress*, Anaheim, CA, USA, November 13–19, 2004.
- [51] Bohm, M. R. and Stone, R. B., 2004, "Representing Functionality to Support Reuse: Conceptual and Supporting Functions," *ASME 2004 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Salt Lake City, UT, USA, September 28 – October 2, 2004.
- [52] Bohm, M. R., Stone, R. B., Simpson, T. W., and Steva, E. D., 2006, "Introduction of a Data Schema: The Inner Workings of a Design Repository," *ASME 2006 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Philadelphia, PA, USA, September 10-13, 2006.
- [53] Bohm, M. R., Stone, R. B., and Szykman, S., 2005, "Enhancing Virtual Product Representations for Advanced Design Repository Systems," *Journal of Computing and Information Science in Engineering*, **5**(4) pp. 360-72.
- [54] Freeman, P. and Newell, A., 1971, "A Model for Functional Reasoning in Design," *International Joint Conference for Artificial Intelligence*, London, UK, Sep 1-3, 1971.

- [55] Simon, H. A., 1996, *The Sciences of the Artificial* The MIT Press. Cambridge, MA.
- [56] Simon, H. A., 1979, *Models of Thought* Yale University Press. New Haven.
- [57] Eastman, C. M., 1969, "Cognitive Processes and Ill-Defined Problems: A Case Study from Design," *First Joint Interantional Confernce on Artificial Intelligence*, Washington, DC, 1969.
- [58] Kroes, P., 2010, "Formalization of Technical Functions: Why Is That So Difficult?," *Tools and Methods of Competitive Engineering, TMCE-2010*, Ancona, Italy, August 12-16, 2010.
- [59] Umeda, Y., Takeda, H., Tomiyama, T., and Yoshikawa, H., "Function, Behavior, and Structure," in *Applications of Artificial Intelligence V, Vol 1: Design*, J. S. Gero, Ed. Boston, MA: Springer Verlag, , 1990, pp. 177-193.
- [60] Chandrasekaran, B. and Josephson, J. R., 1997, "Representing Function as Effect," *Fifth International Workshop on Advances in Functional Modeling of Complex Technical Systems*, Paris, France, July 1997.
- [61] Chandrasekaran, B. and Josephson, J. R., 2000, "Function in Device Representation," *Engineering with Computers*, **16**(3-4) pp. 162-177.
- [62] Chandrasekaran, B., 2005, "Representing Function: Relating Functional Representation and Functional Modeling Research Streams," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **19**(2) pp. 65-74.
- [63] Eck, D. V., McAdams, D. A., and Vermaas, P. E., 2007, "Functional Decomposition in Engineering: A Survey," *ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Las Vegas, NV, USA.
- [64] Sasajima, M., Kitamura, Y., Ikeda, M., and Mizoguchi, R., 1995, "FBRL:A Function and Behavior Representation Language," *International Joint*

- Conferences on Artificial Intelligence*, Montreal, Quebec, Canada, August 20-25, 1995.
- [65] Garbacz, P., 2005-2006, "Towards a Standard Taxonomy of Artifact Functions," *Applied Ontology*, **1**pp. 221–236.
- [66] Cebrian-Tarrason, D., Lopez-Montero, J. A., and Vidal, R., 2008, "Ontofabes: Ontology Design Based in FBS Framework," *CIRP Design Conference 2008: Design Synthesis*, Enschede, Netherlands, April 7-9, 2008.
- [67] Bracewell, R. H. and Sharpe, J. E. E., 1996, "Functional Descriptions Used in Computer Support for Qualitative Scheme Generation—"Schemebuilder",  
*Artificial Intelligence for Engineering Design, Analysis & Manufacturing*, **10**(4) pp. 333-345.
- [68] Goel, A., Bhatta, S., and Stroulia, E., "Kritik: An Early Case-Based Design System," in *Issues and Applications of Case-Based Reasoning in Design*, M. L. Maher and P. Pu, Eds. Mahwah, NJ: Erlbaum, 1997, pp. 87-132.
- [69] Bobrow, D. G., 1984, "Qualitative Reasoning About Physical Systems: An Introduction," *Artificial Intelligence*, **24**(1-3) pp. 1-5.
- [70] Dorst, K. and Vermaas, P. E., 2005, "John Gero's Function-Behaviour-Structure Model of Designing: A Critical Analysis," *Research in Engineering Design*, **16**pp. 17-26.
- [71] Erden, M. S., Komoto, H., VanBeeK, T. J., D'Amelio, V., Echavarria, E., and Tomiyama, T., 2008, "A Review of Function Modeling: Approaches and Applications," *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **22**(2) pp. 147-169.
- [72] Bhatta, S. R. and Goel, A. K., 1997, "A Functional Theory of Design Patterns," *15th International Joint Conference on Artificial Intelligence - Volume 1*, Nagoya, Japan.

- [73] Bhatta, S., Goel, A., and Prabhakar, S., 1994, "Innovation in Analogical Design: A Model-Based Approach," *Artificial Intelligence in Design*, Dordrecht, The Netherlands, 1994.
- [74] Chakrabarti, A. and Bligh, T. P., 1994, "An Approach to Functional Synthesis of Solutions in Mechanical Conceptual Design. Part I: Introduction and Knowledge Representation," *Research in Engineering Design*, **6**(3) pp. 127-141.
- [75] Chakrabarti, A. and Bligh, T. P., 2001, "A Scheme for Functional Reasoning in Conceptual Design," *Design Studies*, **22**pp. 493-517.
- [76] Chakrabarti, A. and Bligh, T. P., 1996, "An Approach to Functional Synthesis of Mechanical Design Concepts: Theory, Applications, and Emerging Research Issues," *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, **10**(4) pp. 313-331.
- [77] Sembugamoorthy, V. and Chandrasekaran, B., "Functional Representation of Devices and Compilation of Diagnostic Problem-Solving Systems," in *Experience, Memory, and Reasoning*, J. Kolodner and C. K. Riesbeck, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, 1986, pp. 47-53.
- [78] Keuneke, A. M., 1991, "Device Representation - the Significance of Functional Knowledge," *IEEE Expert* **6**(2) pp. 22-25.
- [79] Deng, Y. M., 2002, "Function and Behavior Representation in Conceptual Mechanical Design," *Artificial Intelligence in Engineering Design, Analysis, & Manufacturing*, **16**(5) pp. 343-362.
- [80] Zhang, W. Y., Tor, S. B., Britton, G. A., and Deng, Y. M., "Functional Design of Mechanical Products Based on Behavior-Driven Function-Environment-Structure Modeling Framework,"
- [81] Rodenacker, W., 1971, *Methodisches Konstruieren* Springer-Verlag. Berlin.
- [82] Fenves, S. J., 2002, "A Core Product Model for Representing Design Information," *National Institute of Standards & Technology Report No. NISTIR 7185*, Gaithersburg, MD, USA.

- [83] Fenves, S. J., Foufou, S., Bock, C., Sudarsan, R., and Sriram, R. D., 2006, "Cpm: A Core Product Model for Plm Support," *Frontiers in Design Simulation and Research*, Alanta, Georgia, USA.
- [84] Bryant, C. R., McAdams, D. A., and Stone, R. B., 2006, "A Validation Study of an Automated Concept Generator Design Tool," *ASME 2006 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Philadelphia, PA, USA, September 10-13, 2006.
- [85] Caldwell, B. W. and Mocko, G. M., 2007, "Towards Rules for Functional Composition," *ASME 2008 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference*, Brooklyn, NY, USA, August 3-6, 2008.
- [86] Kurfman, M. A., Stone, R. B., Rajan, J. R., and Wood, K. L., 2001, "Functional Modeling Experimental Studies," *ASME Design Engineering Technical Conferences*, Pittsburgh, Pennsylvania, Sep 9-12, 2001.
- [87] Kurfman, M. A., Stone, R. B., Wie, M. V., Wood, K. L., and Otto, K. N., 2000, "Theoretical Underpinnings of Functional Modeling: Preliminary Experimental Studies," *2000 ASME Design Engineering Technical Conference*, Baltimore, MD, USA, Sep 10-13, 2000.
- [88] McAdams, D. A. and Wood, K., 2002, "A Quantitative Similarity Metric for Design-by-Analogy," *Journal of Mechanical Design*, **124**(2) pp. 173-182.
- [89] Collins, J. A., Hagan, B. T., and Bratt, H. M., 1976, "Failure-Experience Matrix - a Useful Design Tool," *Journal of Engineering for Industry*, **B 98**(3) pp. 1074-1079.
- [90] Kirschman, C. F. and Fadel, G. M., 1998, "Classifying Functions for Mechanical Design," *Journal of Mechanical Design*, **120**(3) pp. 475-482.
- [91] Vucovich, J., Bhardwaj, N., Ho, H. H., Ramakrishna, M., Thakur, M., and Stone, R., 2006, "Concept Generation Algorithms for Repository-Based Early Design," *ASME 2006 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Philadelphia, PA, USA, September 10-13, 2006.

- [92] Anandan, S., "Similarity Metrics Applied to Graph Based Design Model Authoring," in *Department of Mechanical Engineering*. vol. PhD Clemson: Clemson University, 2008.
- [93] Anandan, S., Summers, J. D., Maier, J. R. A., Bapat, V., and Bettig, B., 2007, "Semantics in Engineering Design," *International Conference on Engineering Design*, Paris, France.
- [94] Stone, R. B., Tumer, I. Y., and Stock, M. E., 2005, "Linking Product Functionality to Historic Failures to Improve Failure Analysis in Design," *Research in Engineering Design*, **16**(2) pp. 96-108.
- [95] Tumer, I. Y. and Stone, R. B., 2001, "Analytical Methods to Evaluate Failure Potential During High-Risk Component Development," *2001 ASME Design Engineering Technical Conferences*, Pittsburgh, PA, USA, September 9-12, 2001.
- [96] Arunajadai, S. G., Stone, R. B., and Tumer, I. Y., 2002, "A Framework for Creating a Function-Based Design Tool for Failure Mode Identification," *ASME 2002 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Montreal, Canada, September 29 – October 2, 2002.
- [97] Arunajadai, S. G., Uder, S. J., Stone, R. B., and Tumer, I. Y., 2004, "Failure Mode Identification through Clustering Analysis," *Quality and Reliability Engineering International*, **20**pp. 511-526.
- [98] Kurtoglu, T., Campbell, M. I., Bryant, C. R., Stone, R. B., and McAdams, D. A., 2005, "Deriving a Component Basis for Computational Functional Synthesis," *International Conference on Engineering Design, ICED '05*, Melbourne, Australia, Aug 15-18, 2005.
- [99] Baya, V. and Leifer, L., 1995, "Understanding Design Information Handling Behavior Using Time and Information Measure," *ASME Design Engineering Technical Conferences*, Boston, MA.
- [100] Iyer, N., Jayanti, S., Lou, K., Kalyanaraman, Y., and Ramani, K., 2005, "Three-Dimensional Shape Searching: State-of-the-Art Review and Future Trends," *Computer-Aided Design & Applications*, **37**pp. 509-530.



- [101] Cagan, J., Campbell, M. I., and Finger, S., 2005, "A Framework for Computational Design Synthesis: Model and Applications," *Journal of Computing and Information Science in Engineering*, **5**(3) pp. 171-181.
- [102] Linz, P., 1996, *An Introduction to Formal Languages and Automata*, Second ed. D. C. Heath and Company. Lexington, MA, USA.
- [103] Collins, J. A., busby, H. R., and Staab, G. H., 2010, *Mechanical Design of Machine Elements and Machines*, Second ed. John Wiley & Sons. Hoboken, NJ.
- [104] Kurfman, M. A., Stone, R. B., Rajan, J., and Wood, K. L., 2003, "Experimental Studies Assessing the Repeatability of a Functional Modeling Derivational Method," *Journal of Mechanical Design*, **125**pp. 682-693.
- [105] Vermaas, P. E., 2007, "The Functional Modeling Account of Stone and Wood: Some Critical Remarks," *International Conference on Engineering Design, ICED* Cité des Sciences et de L'industrie, Paris, France, Aug 28-31, 2007.
- [106] Lind, M., 1994, "Modeling Goals and Functions of Complex Plant," *Applied Artificial Intelligence*,
- [107] Russell, S. and Norvig, P., 2002, *Artificial Intelligence: A Modern Approach* Prentice Hall/Pearson Education. Upper Saddle River, N.J.
- [108] Chomsky, N., 1956, "Three Models for the Description of Language," *IRE Transactions on Information Theory*, **2**(3) pp. 113-124.
- [109] Russell, S. and Norvig, P., 2003, *Artificial Intelligence: A Modern Approach* Prentice Hall/Pearson Education. Upper Saddle River, N.J.
- [110] Chen, P. P., "A Preliminary Framework for Entity-Relationship Models," in *Second International Conference on the Entity-Relationship Approach to Information Modeling and Analysis*, P. P. Chen, Ed.: North-Holland Publishing Co., 1981, pp. 19-28.

- [111] Luger, G. F., 2002, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, Fourth ed. Addison-Wesley. Essex, England.
- [112] Kleene, S., 1991, *Introduction to Metamathematics* Amsterdam, New York.
- [113] Reichenbach, H., 1947, *Elements of Symbolic Logic* Dover Publications Inc. New York.
- [114] Tarski, A., 1946, *Introduction to Logic and to the Methodology of Deductive Sciences*, Second ed. Dover Publications, Inc. New York.
- [115] Sen, C., Summers, J. D., and Mocko, G. M., 2011, "Exploring Potentials for Conservational Reasoning Using Topologic Rules of Function Structure Graphs," *The 18th International Conference on Engineering Design*, Copenhagen, August 15-18, 2011.
- [116] Sen, C., Summers, J. D., and Mocko, G. M., 2010, "Topological Information Content and Expressiveness of Function Models in Mechanical Design," *Journal of Computing and Information Science in Engineering*, **10**(3) pp. 031003-1 – 031003-11.
- [117] Trucco, E., 1956, "A Note on the Information Content of Graphs," *Bulltin of Mathematical Biophysics*, **18**
- [118] Hazelrigg, G. A., 1999, "On the Role and Use of Mathematical Models in Engineering Design," *Journal of Mechanical Design*, **121**(3) pp. 336-341.
- [119] Baader, F., 1996, "A Formal Definition for the Expressive Power of Terminological Knowledge Representation Languages," *Journal of Logic and Computation*, **6**(1) pp. 33-54.
- [120] Winston, P. H., 1992, *Artificial Intelligence*, Third ed. Addison-Wesley Publishing Company. Reading. Massachusetts, USA.
- [121] Rich, E. and Knight, K., 1991, *Artificial Intelligence*, Second ed. McGraw-Hill, Inc. New York, NY, USA.

- [122] Boolos, G., 1989, "A New Proof of the Gödel Incompleteness Theorem," *Notices of the American Mathematical Society* **36**pp. 388-390 and 676.
- [123] Lienhardt, P., 1991, "Topological Models for Boundary Representation: A Comparison with N-Dimensional Generalized Maps," *Computer-Aided Design & Applications*, **23**(1) pp. 59-82.
- [124] Chavali, S., "A Case Study Investigating Rule Based Design in an Industrial Setting," in *Department of Mechanical Engineering*. vol. MS Clemson, SC: Clemson University, 2007.
- [125] Chavali, S. R., Sen, C., Mocko, G. M., and Summers, J. D., 2008, "Using Rule Based Design in Engineer to Order Industry: An Sme Case Study," *Computer-Aided Design & Applications*, **5**(1-4) pp. 178-193.
- [126] Halliday, D., Resnick, R., and Walker, J., 2011, *Fundamentals of Physics Extended, Ninth Edition*, Ninth ed. John Wiley & Sons, Inc. New York.
- [127] Griffiths, D. J., 1999, *Introduction to Electrodynamics*, Third ed. Prentice-Hall, Inc. Upper Saddle River, NJ.
- [128] Bejan, A., 2006, *Advanced Engineering Thermodynamics*, Third ed. John Wiley & Sons, Inc. Hoboken, NJ.
- [129] Moran, M. J., Shapiro, H. N., Boettner, D. D., and Bailey, M., 2010, *Fundamentals of Engineering Thermodynamics* John Wiley & Sons, Inc. Hoboken, NJ.
- [130] Feynman, R. P., 1985, *Q.E.D. Quantum Electro-Dynamics: The Strange Theory of Light and Matter* Princeton University Press. Princeton, New Jersey, USA.
- [131] Feynman, R. P., 1997, *Six Not-So-Easy Pieces: Einstein's Relativity, Symmetry, and Space-Time* Penguin Putnam Inc. New York City, NY, USA.
- [132] Paredis, C. J., Bernard, Y., Burkhart, R. M., Koning, H. D., Friedenthal, S., Fritzon, P., Rouquette, N. F., and Schamai, W., 2010 "An Overview of the

- SysML-Modelica Transformation Specification," *INCOSE International Symposium*, July 2010.
- [133] Schamai, W., Pohlmann, U., Fritzson, P., Paredis, C. J., Helle, P., and Strobel, C., 2010, *Execution of Umlstate Machines Using Modelica*.
- [134] Davies, K. L., Haynes, C. L., and Paredis, C. J., 2009, "Modeling Reaction and Diffusion Processes of Fuel Cells within Modelica," *7th International Modelica Conference*, Linköping, Sweden, September 2009.
- [135] Vargas-Hernandez, N. and Shah, J. J., 2004, "2nd-Cad: A Tool for Conceptual Systems Design in Electromechanical Domain," *Journal of Computing and Information Science in Engineering*, **4**pp. 28-36.
- [136] Phillips, A. C., "Introduction to Quantum Mechanics," Second ed New York: John Wiley & Sons. Inc., 2003.
- [137] Maier, J., Anandan, S., Bapat, V., Summers, J. D., and Bettig, B., 2007, "A Computational Framework for Semantically Rich Design Problems Based on the Theory of Affordances and Exemplar Technology," *International Conference for Engineering Design*, Paris, France.
- [138] Maier, J. R. A., 2008, "Rethinking Design Thoery," *Mechanical Engineering*, **130**(9) pp. 34-37.
- [139] Maier, J. R. A. and Fadel, G. M., 2001, "Affordance: The Fundamental Concept in Engineering Design," *2001 ASME Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Pittsburgh, Pennsylvania, USA, September 9-12, 2001.
- [140] Maier, J. R. R. and Fadel, G. M., 2002, "Comapring Function and Affordance as Bases for Design," *Proceedings of ASME 2002 Design Engineering Technical Conferences and Computer and Information in Engineering Conference*, Montreal, Canada, Sep 29 - Oct 2, 2002.

- [141] Albers, A., Matthiesen, S., Thau, S., and Alink, T., 2008, "Support of Design Engineering Activity through C&Cm - Temporal Decomposition of Design Problems," *TMCE 2008 Symposium*, Izmir, Turkey, April 21-25, 2008.
- [142] Albers, A., Burkardt, N., and Ohmer, M., 2004, "Principles for Design on the Abstract Level of the Contact & Channel Model," *TMCE 2004*, Lausanne, Switzerland, April 13-17, 2004.
- [143] Umeda, Y., Kondoh, S., Shimomura, Y., and Tomiyama, T., 2005, "Development of Design Methodology for Upgradable Products Based on Function–Behavior–State Modeling," *Artificial Intelligence for Engineering Design, Analysis & Manufacturing*, **19**pp. 161-182.
- [144] Stone, R. B., Tumer, I. Y., and Wie, M. V., 2005, "The Function-Failure Design Method," *Journal of Mechanical Design*, **127**(3) p. 397 (11 pages).
- [145] Forbus, K. D., 1984, "Qualitative Process Theory," *Artificial Intelligence*, **24**pp. 85-168.
- [146] DeKleer, J. and Brown, J. S., 1984, "A Qualitative Physics Based on Confluences," *Artificial Aintelligence*, **24**(1-3) pp. 7-83.
- [147] DeKleer, J. and Brown, J. S., 1984, "A Framework for Qualitative Physics," *Sixth Annual Conference of the Cognitive Science Society*.
- [148] Sim, S. K. and Duffy, A. H. B., 2003, "Towards an Ontology of Generic Engineering Design Activities," *Research in Engineering Design*, **14**pp. 200-223.
- [149] Kurtoglu, T. and Tumer, I. Y., 2008, "A Graph-Based Fault Identification and Propagation Framework for Functional Design of Complex Systems," *ASME Journal of Mechanical Design*, **130**pp. 051401-1 - 051401-8.
- [150] Bondi, A. B., 2000, "Characteristics of Scalability and Their Impact on Performance," *2nd international workshop on Software and performance*, Ottawa, Ontario, Canada.

- [151] Mathieson, J. L., Sen, C., and Summers, J. D., 2009, "Information Generation in the Design Process," *ASME 2009 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, San Diego, CA, USA, August 30 - September 2, 2009.
- [152] Mathieson, J., Shanthakumar, A., Sen, C., Summers, J. D., and Stone, R., 2011, "Complexity as a Surrogate Mapping between Function Models and Market Value," *ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Washington, DC, September 2011.
- [153] Ameri, F., Summers, J. D., Mocko, G. M., and Porter, M., 2008, "Engineering Design Complexity: An Experimental Study of Methods and Measures," *Research in Engineering Design*, **19**(2-3) pp. 161-179.
- [154] Summers, J. D. and Ameri, F., 2008, "An Algorithm for Assessing Design Complexity through a Connectivity View," *TMCE 2008*, Izmir, Turkey, April 20-25, 2008.
- [155] Kayyar, M., Summers, J. D., Ameri, F., and Biggers, S., "A Case Study of Sme Design Process and Development of a Design Enabling Tool," in *ASME Design Engineering Technical Conferences and Computers in Engineering Conferences*. vol. DAC Las Vegas, NV: ASME, 2007, p. #35610.
- [156] Pehlivan, S., Ameri, F., and Summers, J. D., 2009, "An Agent-Based System Approach to Fixture Design," *International Journal of Computer Applications in Technology*, **36**(3-4) pp. 284-296.
- [157] Roth, K., 2000, *Konstruieren Mit Konstruktionskatalogen*, Third ed. Springer Berlin.
- [158] Koller, R., 1998, *Konstruktionslehre Für Den Maschinenbau*, Fourth ed. Springer Berlin.
- [159] Caldwell, B. W., Sen, C., Mocko, G. M., and Summers, J. D., 2011, "An Empirical Study of the Expressiveness of the Functional Basis," *Artificial Intelligence for Engineering Design, Analysis & Manufacturing* **25**pp. 273-287.

- [160] Merrie, B., Moor, J., and Nelson, J., 2009, *The Logic Book* Fifth ed. McGraw-Hill, Inc. New York.
- [161] Stroustrup, B., 1997, *The C++ Programming Language*, Third ed. Addison-Wesley Publishing Company. Reading, Massachusetts.
- [162] Chen, P. P., 1976, "The Entity-Relationship Model - toward a Unified View of Data," *ACM Transactions on Database Systems*, **1**(1) pp. 9-36.
- [163] Chen, P. P., 1980, "Entity-Relationship Approach to Systems Analysis and Design," *First International Conference on the Entity-Relationship Approach*, North-Holland.
- [164] Chen, P. P., 1981, "Entity-Relationship Approach to Information Modeling and Analysis," *Second International Conference on the Entity-Relationship Approach*, Washington, DC, USA, October 12-14, 1981.
- [165] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., 2003, *The Description Logic Handbook: Theory, Implementation and Applications* Cambridge University Press. Cambridge, UK.
- [166] Caldwell, B. W., Sen, C., Mocko, G. M., Summers, J. D., and Fadel, G. M., 2008, "Empirical Examination of the Functional Basis and Design Repository," *Third International Conference on Design Computing and Cognition*, Atlanta, GA, USA, June 23-25, 2008.
- [167] Prosise, J., 1999, *Programming Windows with Mfc*, Second ed. Microsoft Press. Redmond, WA.
- [168] Jones, R. M., 2000, *Introduction to Mfc Programming with Visual C++* Prentice-Hall, Inc. Upper Saddle River, NJ.
- [169] Harrison, M. A., 1978, *Introduction to Formal Language Theory* Addison-Wesley.

- [170] Mateescu, A. and Salomaa, A., "Aspects of Classical Language Theory," in *Handbook of formal languages*. vol. 1, G. Rozenberg and A. Salomaa, Eds. New York, NY, USA: Springer-Verlag, 1997.
- [171] Hopcroft, J., Motwani, R., and Ullman, J., 2001, *Introduction to Automata Theory, Languages, and Computation*, Second ed. Addison-Wesley. New York, NY, USA.
- [172] Gruber, T., 1993, "A Translation Approach to Portable Ontology Specifications," *Knowledge Acquisition*, **5**(199-220)
- [173] Patil, L., Dutta, D., and Sriram, R., 2005, "Ontology-Based Exchange of Product Data Semantics," *IEEE Transactions on Automation Science and Engineering*, **2**(3) pp. 213-225.
- [174] Li, L. and Horrocks, I., 2004, "A Software Framework for Matchmaking Based on Semantic Web Technology," *International Journal of Electronic Commerce*, **8**(4) pp. 39-60.
- [175] Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., and Katz, Y., 2007, "Pellet: A Practical OWL-DL Reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, **5**(2) pp. 51-53.
- [176] Ma, J., Summers, J. D., and Joseph, P., 2010, "Simulation Studies on the Influence of Obstacles on Rolling Lunar Wheel," *ASME International Design Engineering Technical Conferences*, Montreal, Canada, August, 2010.
- [177] Johnson, T. A., Paredis, C. J., and Burkhart, R. M., 2008, "Integrating Models and Simulations of Continuous Dynamics into SysML," *6th International Modelica Conference*, Linköping, Sweden.
- [178] Schultz, J., Sen, C., Mathieson, J., Caldwell, B. W., Summers, J. D., and Mocko, G. M., 2010, "Limitations to Function Structures: A Case Study in Morphing Airfoil Design," *ASME International Design Engineering Technical Conferences, DTM*, Montreal, Canada, August, 2010.



- [179] Caldwell, B. W. and Mocko, G. M., 2011, "Validation of Function Pruning Rules through Similarity at Three Levels of Abstraction," *Journal of Mechanical Design (Accepted)*,
- [180] Zeid, I., 2007, *Mastering Cad/Cam*, First ed. Tata McGraw-Hill Publishing Company Limited. New Delhi, India.
- [181] Ramachandran, R., Caldwell, B. W., and Mocko, G. M., 2011, "A User Study to Evaluate the Function Model and Function Interaction Model for Concept Generation," *ASME International Design Engineering Technical Conferences*, Washington, DC, September, 2011.
- [182] Nagel, R. L., "A Design Framework for Identifying Automation Opportunities," in *School of Mechanical, Industrial and Manufacturing Engineering*. vol. PhD Corvallis, OR: Oregon State University, 2011.
- [183] Nagel, R. L., Bohm, M. R., Stone, R. B., and McAdams, D. A., "A Representation of Carrier Flows for Functional Design," *International Conference on Engineering Design ICED 07*, Paris, France, August 28-31, 2007.
- [184] Cardella, M. E., Atman, C. J., and Adams, R. S., 2006, "Mapping between Design Activities and External Representations for Engineering Student Designers," *Design Studies*, **27**pp. 5-24.