



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

**IDENTIFICATION OF BEHAVIOR PATTERNS
IN SYSTEMS-OF-SYSTEMS ARCHITECTURES**

by

John J. Quartuccio

March 2020

Dissertation Supervisor:

Kristin M. Giammarco

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE March 2020	3. REPORT TYPE AND DATES COVERED Dissertation	
4. TITLE AND SUBTITLE IDENTIFICATION OF BEHAVIOR PATTERNS IN SYSTEMS-OF-SYSTEMS ARCHITECTURES			5. FUNDING NUMBERS	
6. AUTHOR(S) John J. Quartuccio				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) This dissertation presents a methodology to derive inherent behavior patterns in system-of-systems architectures. When considering the possibility of not only the intended behaviors of a system but also the alternative, unintended behaviors of the system, the developer needs to evaluate hundreds of possible outcomes, even for a simple model. This set of outcomes grows to tens of thousands for a reasonably complex model, driving the need for some means of systematic analysis that ensures the evaluation of all possibilities. This dissertation outlines a methodology that enables the developer to describe the behaviors of a particular system; define the interaction of the system with other systems, users, and the environment; add constraints to the model; derive patterns from the set of possible outcomes; analyze and interpret the results; and re-use the model for other problems of interest. The methodology employs the Monterey Phoenix behavior modeling concepts and tools as a light-weight, formal method. The dissertation details a cross-domain example that demonstrates the methodology and outlines multiple applications, serving as verification of the approach.				
14. SUBJECT TERMS pattern identification, behavior model, semantic search, small-scope hypothesis, model-based systems engineering, MBSE, Monterey Phoenix, SysML, UML, DoDAF, architecture, activity diagram, sequence diagram, N-squared diagram, topology, ontology, Bayes network, directed acyclic graph, isomorphic systems process, system of systems, general systems theory, ontology design pattern, design structure matrix			15. NUMBER OF PAGES 445	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

**IDENTIFICATION OF BEHAVIOR PATTERNS IN SYSTEMS-OF-SYSTEMS
ARCHITECTURES**

John J. Quartuccio
Civilian, Department of the Navy
BS, Pennsylvania State University, 1985
MS, Lehigh University, 1997

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SYSTEMS ENGINEERING

from the

**NAVAL POSTGRADUATE SCHOOL
March 2020**

Approved by: Kristin M. Giammarco
Department of
Systems Engineering
Dissertation Supervisor

Mikhail Auguston
Department of
Computer Science

Ronald E. Giachetti
Department of
Systems Engineering

Raymond J. Madachy
Department of
Systems Engineering

Clifford A. Whitcomb
Department of
Systems Engineering

Kristin M. Giammarco
Department of
Systems Engineering
Dissertation Chair

Approved by: Ronald E. Giachetti
Chair, Department of Systems Engineering

Orrin D. Moses
Vice Provost of Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This dissertation presents a methodology to derive inherent behavior patterns in system-of-systems architectures. When considering the possibility of not only the intended behaviors of a system but also the alternative, unintended behaviors of the system, the developer needs to evaluate hundreds of possible outcomes, even for a simple model. This set of outcomes grows to tens of thousands for a reasonably complex model, driving the need for some means of systematic analysis that ensures the evaluation of all possibilities. This dissertation outlines a methodology that enables the developer to describe the behaviors of a particular system; define the interaction of the system with other systems, users, and the environment; add constraints to the model; derive patterns from the set of possible outcomes; analyze and interpret the results; and re-use the model for other problems of interest. The methodology employs the Monterey Phoenix behavior modeling concepts and tools as a light-weight, formal method. The dissertation details a cross-domain example that demonstrates the methodology and outlines multiple applications, serving as verification of the approach.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1	Introduction	1
1.1	Background	3
1.1.1	Behavior Model Concepts within a System of Systems (SoS).	3
1.1.2	Enabling Technologies for Designed Systems of Systems	5
1.1.3	Potential Sources of Unfavorable Behavior	5
1.2	Statement of the Problem	7
1.3	Research Objective	8
1.4	Purpose of the Research	8
1.5	Theoretical Framework	9
1.6	Scope of Research	10
1.7	Assumptions and Limitations of the Study	10
2	Prior Work	13
2.1	Review of Literature	13
2.1.1	Systems Science	16
2.1.2	Systems Engineering (SE)	21
2.1.3	Modeling and Simulation	26
2.1.4	Model-Based Systems Engineering (MBSE)	30
2.1.5	Ontology	37
2.1.6	Formal Methods	43
2.1.7	Digital Engineering (DE)	45
2.1.8	Systems Engineering (SE) Resources	46
2.2	Assessment of Previous Work	47
2.2.1	Engineering Context.	48
2.2.2	Challenges of the MBSE Vision	48
2.2.3	Ontologies and Semantics	49
2.2.4	Patterns and Reuse	49
2.2.5	Pattern Concepts Applied to SoS Behaviors	49
2.3	Definition of Terms	49
2.3.1	Systems.	49
2.3.2	Models	51
2.3.3	Behavior	55
2.3.4	System Processes and Patterns	60
2.3.5	Summary of Terms	62

3	Research Method	65
3.1	Research Design	65
3.1.1	Pattern Detection	65
3.2	Data Sources, Collection, and Analysis.	66
3.3	Research Concept	67
3.3.1	Pattern Search Strategy.	68
3.3.2	Elements of Behavior Patterns	68
3.3.3	Scope of Execution	73
3.3.4	System Behaviors Compiled as a Directed Graph	74
3.4	Dissertation Contribution	77
3.4.1	Templates Applied to Behavior Models	79
4	Research Findings	85
4.1	Methodology for Pattern Derivation	85
4.1.1	Implementing Monterey Phoenix (MP) within the Methodology	93
4.1.2	Methodology Description.	95
4.1.3	Model Re-use	106
4.2	Derivation of Post-execution Analysis	111
4.2.1	Probability of Each Trace	111
4.2.2	The Design Structure Matrix (DSM)	125
4.3	Verification of the Methodology	128
4.3.1	Derivation of the Decision Model	128
4.3.2	Derivation of the Object-to-Effect Model	173
4.4	Summary of Findings	231
5	Summary	233
5.1	Discussion	233
5.2	Conclusions	235
5.3	Suggestions for Further Research	235
5.3.1	Recommended Capabilities for Monterey Phoenix (MP)	236
5.3.2	Pattern Repository and Management	238
Appendix A	Behavior Models	239
A.1	Decision Model	239
A.1.1	Decision Model Graphic Output	239
A.1.2	Decision Model MP Code Listing	246

A.2	Object-to-Effect Model	267
A.2.1	Object-to-Effect Model Graphic Output	267
A.2.2	Object-to-Effect Model: MP Code Listing	281
Appendix B Probability Calculations		315
B.1	Probability Calculation of the Example Problem	316
B.2	Probability Calculation of the Decision Model	322
B.3	Probability Calculation of the Object-to-Effect Model	333
Appendix C Derivation of the Design Structure Matrix (DSM)		345
C.1	Decision Model	346
C.1.1	DSM Derived from the Decision Model	346
C.1.2	Comparison of Decision Model Results at Scope 1 to 2	356
C.1.3	Cluster Analysis of the Decision Model	358
C.2	Object-to-Effect Model	364
C.2.1	DSM Derived from the Object-to-Effect Model	364
C.2.2	Comparison of Object-to-Effect Models at Scope (1111) and (2111)	374
C.2.3	Cluster Analysis of the Object-to-Effect Model	384
List of References		391
Initial Distribution List		407

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

Figure 2.1	Systems engineering process	22
Figure 2.2	Example of a simple Markov chain	25
Figure 2.3	SysML taxonomy.	34
Figure 2.4	Representation of the Unified Architecture Framework (UAF) Profile . .	36
Figure 2.5	Hierarchy of Basic Formal Ontology (BFO) continuants	38
Figure 2.6	Hierarchy of Basic Formal Ontology (BFO) occurrents	39
Figure 2.7	Pattern meta-model.	42
Figure 2.8	Cost of implementing systems engineering patterns.	43
Figure 3.1	Identifying common patterns among traces.	67
Figure 3.2	MP diagram key.	70
Figure 3.3	Behavior pattern and template definition	71
Figure 3.4	Behavior pattern example	72
Figure 3.5	Examples of the pattern met	73
Figure 3.6	Scope of Execution Example.	74
Figure 3.7	Conceptual example of patterns	76
Figure 3.8	Simple example of patterns	77
Figure 3.9	Decision Model template abstraction.	81
Figure 3.10	Decision Model domain-specific applications.	82
Figure 3.11	Decision Model with two subordinates.	83
Figure 3.12	Decision Model identifying both positive and negative events.	84
Figure 4.1	A methodology to develop the SoS behavior architecture	87
Figure 4.2	Reuse concept.	92

Figure 4.3	Observe, Orient, Decide, Act (OODA) loop segment.	92
Figure 4.4	Coordinated OODA loop segment.	93
Figure 4.5	The Monterey Phoenix (MP) model environment.	94
Figure 4.6	A simple Monterey Phoenix (MP) model.	96
Figure 4.7	A simple Monterey Phoenix (MP) model with coordination.	98
Figure 4.8	A simple Monterey Phoenix (MP) model with a missing constraint. . . .	100
Figure 4.9	A simple Monterey Phoenix (MP) model with assertion checking.	102
Figure 4.10	Cycle Isomorphic Systems Process (ISP) model.	106
Figure 4.11	Cycle ISP model attributes identified.	107
Figure 4.12	Cycle of human rest period.	109
Figure 4.13	Cycle of expansion and contraction.	110
Figure 4.14	Domains within MP	113
Figure 4.15	Fundamental probability axioms applied to the Monterey Phoenix schema.	113
Figure 4.16	Conditional probability belief network	114
Figure 4.17	Verification of conditional probability calculation	115
Figure 4.18	Drive Car Model	118
Figure 4.19	Conditional probability structure	119
Figure 4.20	A single constraint	120
Figure 4.21	Multiple constraints	120
Figure 4.22	Prorating the belief network.	121
Figure 4.23	Constrained Bayesian Network	122
Figure 4.24	Prorating the constrained probability network	124
Figure 4.25	Example traces	125
Figure 4.26	Example N-squared diagram and Design Structure Matrix (DSM)	126
Figure 4.27	Derivation of the DSM	127

Figure 4.28	Universal and model-type specific templates.	130
Figure 4.29	Decision pattern	131
Figure 4.30	Decision Model events associated with the environment	133
Figure 4.31	Decision Model events associated with the leader	134
Figure 4.32	Decision Model events associated with the subordinate(s)	135
Figure 4.33	Decision Model with multiple subordinates	136
Figure 4.34	Decision Model coordination	137
Figure 4.35	Functional Flow Block Diagram (FFBD) of the Decision Model.	139
Figure 4.36	Decision Model constraints	140
Figure 4.37	The effect of constraints on the Decision Model.	143
Figure 4.38	Decision Model attributes aligned to the environment	146
Figure 4.39	Decision Model attributes aligned to the leader	146
Figure 4.40	Decision Model attributes aligned to each subordinate	147
Figure 4.41	Decision Model topology	148
Figure 4.42	Decision Model identifying positive and negative events	151
Figure 4.43	Decision Model capturing transmit and receive segments	152
Figure 4.44	Decision Model, capturing the OODA segment	154
Figure 4.45	Additional subordinate with no supporting template.	156
Figure 4.46	Additional subordinate with Template 13	157
Figure 4.47	Additional subordinate with Template 14	157
Figure 4.48	Decision Model alternative events	160
Figure 4.49	Decision Model probability network	161
Figure 4.50	Decision Model cluster algorithm results.	168
Figure 4.51	Decision Model surgical application	169
Figure 4.52	Decision Model aviation mishap application	171

Figure 4.53	Universal and model-type specific templates.	174
Figure 4.54	Object-to-Effect Model.	175
Figure 4.55	Object-to-Effect Model environment events.	176
Figure 4.56	Object-to-Effect Model object events.	177
Figure 4.57	Object-to-Effect Model multiple objects.	178
Figure 4.58	Object-to-Effect Model sensor events.	179
Figure 4.59	Object-to-Effect Model multiple sensor events.	179
Figure 4.60	Object-to-Effect Model processor events.	180
Figure 4.61	Object-to-Effect Model multiple processor events.	181
Figure 4.62	Object-to-Effect Model actor events.	182
Figure 4.63	Object-to-Effect Model multiple actors.	182
Figure 4.64	Object-to-Effect Model interaction of the environment with the objects. .	183
Figure 4.65	Object-to-Effect Model interaction of objects with sensors.	185
Figure 4.66	Object-to-Effect Model interaction of sensors with processors.	187
Figure 4.67	Object-to-Effect Model interaction of processors with actors.	189
Figure 4.68	Object-to-Effect Model interaction of actors with objects.	191
Figure 4.69	FFBD of the Object-to-Effect Model.	192
Figure 4.70	Constraints applied to the Object-to-Effect Model.	194
Figure 4.71	Object-to-Effect Model environmental attributes.	200
Figure 4.72	Object-to-Effect Model object attributes.	201
Figure 4.73	Object-to-Effect Model sensor attributes.	203
Figure 4.74	Object-to-Effect Model processor attributes.	204
Figure 4.75	Object-to-Effect Model actor attributes.	205
Figure 4.76	Object-to-Effect Model topology with attributes.	207
Figure 4.77	Object-to-Effect Model Template 9.	210

Figure 4.78	Object-to-Effect Model analysis: trace 35 of 72	213
Figure 4.79	Object-to-Effect Model analysis: trace 35 of 72	214
Figure 4.80	Object-to-Effect Model analysis: trace 32 of 70.	216
Figure 4.81	Object-to-Effect Model analysis: trace 28 of 66.	217
Figure 4.82	Object-to-Effect Model analysis: trace 15 of 30	218
Figure 4.83	Object-to-Effect Model analysis: trace 54 of 130	219
Figure 4.84	Object-to-Effect Model probability.	220
Figure 4.85	Object-to-Effect Model probability belief network.	222
Figure 4.86	Object-to-Effect Model re-use of a SoS.	228
Figure 4.87	Vehicle sensor model.	230
Figure A.1	Decision Model, scope 1, trace 1 of 12.	239
Figure A.2	Decision Model, scope 1, trace 2 of 12.	240
Figure A.3	Decision Model, scope 1, trace 3 of 12.	240
Figure A.4	Decision Model, scope 1, trace 4 of 12.	241
Figure A.5	Decision Model, scope 1, trace 5 of 12.	241
Figure A.6	Decision Model, scope 1, trace 6 of 12.	242
Figure A.7	Decision Model, scope 1, trace 7 of 12.	242
Figure A.8	Decision Model, scope 1, trace 8 of 12.	243
Figure A.9	Decision Model, scope 1, trace 9 of 12.	243
Figure A.10	Decision Model, scope 1, trace 10 of 12.	244
Figure A.11	Decision Model, scope 1, trace 11 of 12.	244
Figure A.12	Decision Model, scope 1, trace 12 of 12.	245
Figure A.13	Object-to-Effect Model, scope (1-1-1-1), trace 1 of 18, Template 1: Not an object of interest.	267
Figure A.14	Object-to-Effect Model, scope (1-1-1-1), trace 2 of 18, Template 2: Not an object of interest, merged data not received.	268

Figure A.15	Object-to-Effect Model, scope (1-1-1-1), trace 3 of 18, Template 3: Not an object of interest, merged data not sent.	268
Figure A.16	Object-to-Effect Model, scope (1-1-1-1), trace 4 of 18, Template 4: Not an object of interest, interpreted as unknown.	269
Figure A.17	Object-to-Effect Model, scope (1-1-1-1), trace 5 of 18, Template 5: Not an object of interest, data not received.	269
Figure A.18	Object-to-Effect Model, scope (1-1-1-1), trace 6 of 18, Template 6: Not an object of interest, data not sent.	270
Figure A.19	Object-to-Effect Model, scope (1-1-1-1), trace 7 of 18, Template 7: Not an object of interest, not detected.	270
Figure A.20	Object-to-Effect Model, scope (1-1-1-1), trace 8 of 18, Template 8: Not an object of interest, not observable.	271
Figure A.21	Object-to-Effect Model, scope (1-1-1-1), trace 9 of 18, Template 9: Object receives effect.	271
Figure A.22	Object-to-Effect Model, scope (1-1-1-1), trace 10 of 18, Template 10: Object acted upon without effect.	272
Figure A.23	Object-to-Effect Model, scope (1-1-1-1), trace 11 of 18, Template 11: Object of interest, merged data received, but no action taken.	272
Figure A.24	Object-to-Effect Model, scope (1-1-1-1), trace 12 of 18, Template 12: Object of interest, merged data not received.	273
Figure A.25	Object-to-Effect Model, scope (1-1-1-1), trace 13 of 18, Template 13: Object of interest, merged data not sent.	273
Figure A.26	Object-to-Effect Model, scope (1-1-1-1), trace 14 of 18, Template 14: Object of interest, interpreted as unknown.	274
Figure A.27	Object-to-Effect Model, scope (1-1-1-1), trace 15 of 18, Template 15: Object of interest, data not received.	274
Figure A.28	Object-to-Effect Model, scope (1-1-1-1), trace 16 of 18, Template 16: Object of interest, data not sent.	275
Figure A.29	Object-to-Effect Model, scope (1-1-1-1), trace 17 of 18, Template 17: Object of interest, not detected.	275
Figure A.30	Object-to-Effect Model, scope (1-1-1-1), trace 18 of 18, Template 18: Object of interest, not observable.	276

Figure A.31	Object-to-Effect Model, scope (2-1-1-1), trace 35 of 72, Templates 9 and 19: Isolated Object.	277
Figure A.32	Object-to-Effect Model, scope (1-2-1-1), trace 32 of 70, Templates 9 and 20: Isolated Sensor.	278
Figure A.33	Object-to-Effect Model, scope (1-1-2-1), trace 28 of 66, Templates 9 and 21: Isolated Processor.	279
Figure A.34	Object-to-Effect Model, scope (1-1-1-2), trace 15 of 30, Templates 9 and 22: Isolated Actor.	280

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

Table 1.1	Number of alternatives, given two options.	2
Table 2.1	Fundamental systems processes.	18
Table 2.2	System-level processes.	19
Table 2.3	Classifications of emergence.	24
Table 2.4	Types of modeling and simulation.	30
Table 2.5	INCOSE MBSE roadmap.	37
Table 2.6	General pattern definition	41
Table 3.1	Research method	66
Table 3.2	Methodology design attributes	67
Table 4.1	A simple Design Structure Matrix (DSM).	104
Table 4.2	Decision Model constraints	141
Table 4.3	Decision Model attributes	144
Table 4.4	Decision Model run matrix	155
Table 4.5	Decision Model templates.	159
Table 4.6	Decision Model probability	161
Table 4.7	Decision Model probability after attributing goals	163
Table 4.8	The Decision Model DSM, scope of 1	165
Table 4.9	The Decision Model DSM, scope of 2	166
Table 4.10	The Decision Model DSM, scope of 3	167
Table 4.11	Object-to-Effect Model interaction of the environment with the objects. .	184
Table 4.12	Object-to-Effect Model interaction of objects with sensors.	185

Table 4.13	Object-to-Effect Model interaction of sensors with processors.	187
Table 4.14	Object-to-Effect Model interaction of processors with actors.	189
Table 4.15	Object-to-Effect Model interaction of actors with objects.	191
Table 4.16	Object-to-Effect Model constraints 1 and 2.	194
Table 4.17	Object-to-Effect Model constraints 3 and 4.	195
Table 4.18	Object-to-Effect Model constraints 5 through 10.	196
Table 4.19	Object-to-Effect Model constraints 11 through 16.	197
Table 4.20	Object-to-Effect Model functional attributes.	199
Table 4.21	Object-to-Effect Model environmental attributes.	200
Table 4.22	Object-to-Effect Model object attributes.	201
Table 4.23	Object-to-Effect Model communication attributes.	202
Table 4.24	Object-to-Effect Model OODA loop attributes	203
Table 4.25	Object-to-Effect Model templates.	208
Table 4.26	Object-to-Effect run matrix.	211
Table 4.27	Object-to-Effect Model probability calculation.	221
Table 4.28	Object-to-Effect Model outcomes.	222
Table 4.29	Object-to-Effect Model improved outcomes.	223
Table 4.30	Object-to-Effect Model DSM at scope of (1-1-1-1)	225
Table 4.31	Object-to-Effect Model DSM at scope of (2-1-1-1)	226

List of Acronyms and Abbreviations

AFUTL	Air Force Universal Task List
BFO	Basic Formal Ontology
CAS	Complex Adaptive System
CASs	Complex Adaptive Systems
CODeP	Conceptual Ontology Design Pattern
CRUSER	Consortium for Robotics and Unmanned Systems Education and Research
DARPA	Defense Advanced Research Project Agency
DAML	Defense Advanced Research Project Agency (DARPA) Agent Markup Language
DAG	Defense Acquisition Guide
DAG	Directed Acyclic Graph
DAGs	Directed Acyclic Graphs
DAU	Defense Acquisition University
DE	Digital Engineering
DEVS	Discrete Event System Specification
DoD	Department of Defense
DoDAF	Department of Defense Architecture Framework
DSM	Design Structure Matrix
DT	Digital Transformation
DTIC	Defense Technical Information Center
EO/IR	Electro-Optical/Infra-Red
FFBD	Functional Flow Block Diagram
GPS	Global Position System
GST	General Systems Theory
IEEE-CS	Institute of Electrical and Electronics Engineers Computer Society

IDEAS	International Defence Enterprise Architecture Specification Foundation
INCOSE	International Council on Systems Engineering
IC	Inputs in Columns
IR	Inputs in Rows
ISO	International Organization for Standardization
ISP	Isomorphic Systems Process
ISPs	Isomorphic Systems Processes
JMETL	Joint Mission Essential Task List
JP-1	DOD Dictionary of Military and Associated Terms
LiDAR	Light Detection and Ranging
MBSE	Model-Based Systems Engineering
mdk	Model Development Kit
mms	Model Management System
MoDAF	Ministry of Defence Architecture Framework
MP	Monterey Phoenix
M&S	Modeling and Simulation
N2	N-Squared
NAVAIR	Naval Air Systems Command
ODPs	Ontology Design Patterns
OMG	Object Modeling Group
OpenMBEE	Open Model Based Engineering Environment
OODA	Observe, Orient, Decide, Act
OIL	Ontology Inference Layer
OP	Ontology Pattern
OWL	Web Ontology Language
PBSE	Pattern-Based Systems Engineering
RF	Radio Frequency
SE	Systems Engineering

SEBoK	Systems Engineering Body of Knowledge
SET	Systems Engineering Transformation
SERC	Systems Engineering Research Center
SG	Systems Geometry
SME	Subject Matter Expert
SoS	System of Systems
SoSs	Systems of Systems
SPM	SoS Performance Measure
SPT	Systems Processes Theory
SQL	Structured Query Language
SysML	Systems Modeling Language
T&E	Test and Evaluation
TPM	Technical Performance Measure
UAF	Unified Architecture Framework
UARC	University-Affiliated Research Center
UJTL	Universal Joint Task List
UML	Unified Modeling Language
UNJTL	Universal Naval Task List
UPDM	Unified Profile for Department of Defense Architecture Framework (DoDAF) and Ministry of Defence Architecture Framework (MoDAF)
UPIA	Unified Modeling Language (UML) Profile-based Integrated Architecture
ve	View Editor

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

In this dissertation, the author defines a methodology to derive inherent behavior patterns in system architectures; these patterns consist of an observable sequence of events. The developer of a designed system of systems will naturally emphasize the intended path of execution, but the sequences of alternate paths introduces the unexpected patterns, leading to a particular outcome of the behavior that may be either positive or negative. Deriving these patterns of system behavior early in the development cycle enables the user to characterize large sets of potential outcomes of a system, find similarities across multiple domains, and re-use successful architectures. In order to ensure that the system under development accomplishes its intended function and that the system does not exhibit unwanted functions, the developer needs to evaluate multiple paths of execution of the model: (1) those that are expected, or designed-in to the system and (2) those that are also inherent in the system, but not expected by the developer. The system behavior may exhibit both positive and negative patterns [1] that introduce design objectives to be managed by the developer.

The possible sequences of events of a behavior model could include many thousands of different outcomes, based on the level of complexity of the model and the scope of execution needed to derive the expected behaviors. Expanding upon this thought, if a sequence of events includes ten consecutive decisions with two alternatives, then there exists over a thousand ($2^{10} = 1024$) possible outcomes. A reasonably complex system may include thirty decision points, resulting in over a billion outcomes. Two of these systems interacting together would double the decision points to sixty, and the unrestricted results are in the quintillions of sequences. These numbers are too large to without some automated means to identify trends in the results. Otherwise, the developer may be expected to manually inspect on the order of tens or perhaps hundreds of sequences, and so an automated means to inspect behavior model becomes vital. Analysis needs to provide a means to to “detect, classify, predict, control, and visualize behavior in system of systems” [2].

Benefits of identifying patterns in System of Systems (SoS) architectures include the following:

- Enabling a logically robust system architecture, developed interactively with the model that exposes potential flaws,
- Identifying both positive and negative behavior patterns inherent in the system architecture,
- Re-using successful models with the potential benefits of reduced cycle-time to develop new models, reduced labor cost of development, [3], and increased likelihood of developing a successful model.

The following concepts were useful tools for shaping the methodology for detecting, classifying and cataloging behavior patterns:

The concept of an abstraction: An abstraction is disassociated from a particular instance or occurrence. For example, the role of a leader is an abstraction for particular kinds of leaders, such as a surgeon, pilot, professor, or manager. Employing a model at a relatively high level of abstraction enables consideration of the structure of the model without engaging in the level of detail needed for production of the system, such as software coding, physical design, and fabrication.

The concept of an event: Let us consider an event as an abstraction of any particular activity. An event has an observable beginning and ending, with a duration of zero to some positive value of time. To read is an event that requires the participant to engage in the activity over time. The point at which one would lose balance is an instantaneous event, since at one point in time balance is maintained until it is lost.

The concept of relationships among events: There are two fundamental relationships when considering the association of events among other events. *First*, consider the relationship of “inclusion,” such that an event exists within some level of hierarchy. For example, an event of driving a car includes sitting behind the wheel, holding on to the steering wheel, depressing the accelerator, and depressing the brake pedal. Events may also be shared such that a person driving the car and another person walking along the sidewalk may participate the same experience as the car passes the pedestrian. *Second*, consider the relationship of “precedence,” that defines the temporal relationships. Precedence includes the specified ordering as well as the intentional non-specified ordering of events, or concurrency. An example of a precedence relationship is that writing a letter must occur prior to reading that same letter. Conversely, concurrent events may include reading the letter while riding in a train.

These concepts are formalized within the Monterey Phoenix (MP) Firebird tool [4] which was used extensively in order to facilitate the findings of this dissertation. MP affords a means to build behavior, employing light-weight formal methods, models that are adaptable to systems engineering, and extensible to levels of abstraction suitable for the engineering and design of systems of systems [5], [6]. These structures, coded with a formal language, are then executed to form a comprehensive series of traces, each representing a unique series of events.

MP was inspired by the analysis of software structures, specifically by the Small Scope Hypothesis [7] and the associated Alloy Analyzer environment [8]. Consistent with this construct, MP finds all possible outcomes of a behavior model within the scope of execution. The Small Scope Hypothesis

suggests that most problems in the architecture can be found with only a few iterations of the model [7], and so execution of the model affords the developer to be assured that all permutations have been exhausted, within the scope of execution. The execution environment supports assertion checking, trace visualization, annotation of each instance of behavior, and post-execution results in graphical and data file form. These data furthered the contribution of the dissertation.

Methodology: This contribution, a methodology to identify and control behavior patterns of a System of Systems (SoS), is encompassed by the following steps:

1. *Develop a narrative of the behavior.* This step relies heavily upon input of a subject matter expert in concert with the model developer. From the narrative, the developer derives the main actors, the associated events identified by each actor, the sequence of events, and the interactions among the events. Of particular interest is for the subject matter expert to identify alternatives to the expected path of execution.
2. *Identify the events associated with each system.* From the narrative, the developer now formalizes each of the relevant events. This step translates the narrative to the building blocks of the behavior model, establishing the hierarchical (inclusion) and temporal (precedence) relationships each set of events. For an SoS, this step defines the events within each system.
3. *Define the coordination among the events.* This step outlines the temporal (precedence), and user-defined relationships of the events within the model. At the completion of this step, all possible patterns of execution are defined by the model. For an SoS, this step defines the external interaction from system to system.
4. *Derive the constraints of the model.* Given the potential alternatives at each event, execution of the model produces hundreds or even thousands of possible execution paths. The developer uses constraints within the model in order to enable the desirable execution paths and to limit undesirable execution paths. These constraints then become a set of requirements that are needed to produce expected results of the system under development. Types of constraints are listed as follows:
 - *Logical* constraints limit the model from executing fallacious paths, such as receiving a message when none has been sent.
 - *Design* constraints limit the model from executing paths in order to control the desired pattern of the system. For example, a system that incorporates multiple sensors may include a voting scheme based on the desired performance of the system.
 - *Simplification* constraints limit the output by restraining alternate paths that are not of interest. For example, a user may always decide to initiate a sequence, and the model

may eliminate the possibility of not initiating a particular sequence.

- *Definition* constraints limit the output by outlining a pattern such as success-failure criteria. As an example, a correct decision within the model may be considered as a success criteria, and so the constraint eliminates an incorrect decision.

Verification and validation of the model is closely related to the application of the constraints, and so the developer would be expected to conduct many iterations between these steps in order to ensure that the final model produces expected and manageable outcomes.

- *Verification*: In a general sense, as opposed to a more strict definition within Department of Defense (DoD) acquisition [9], verification ensures that the model was constructed in the correct way, and so much of the concentration of effort should be to ensure that the logic inherent in the model makes sense. Initially, most models are prone to have logical inconsistencies. By inspection, the developer needs to ensure that the inconsistencies are removed. Examples of logical inconsistencies abound, and they are dependent upon the context and meaning of the model: a message cannot be received if not sent, a book cannot be read if not written, and a car can not be driven without a driver. The meaning of the event and the context within which it is used make up the *semantics* of the model.
- *Validation*: Also described here in a general sense, validation ensures that the model is useful, such that the model, as an abstraction of the associated end item, accomplishes its intended purpose. And so at this step, interaction between the developer and the Subject Matter Expert (SME) provides an opportunity to ensure that the design constraints are effective, and that the simplifications and definitions are appropriate. Certain deficiencies of the model architecture may come to the forefront at this step, and so sections of the model may need to be re-constructed in order to address deficiencies. For example, if particular paths of execution expose the user to risk of harm (e.g. physical inspection), the developer may propose an approach that completely avoids the risk (e.g. remote inspection).

Key point on constraints:

The constraints establish a formal definition of system requirements, derived interactively with the model.

5. *Identify the patterns within the behavior model.* Ideally, the developer has now constrained the model from thousands of possible outcomes to tens of outcomes that form patterns of possible execution paths of the model that may contain both desirable and undesirable outcomes since it may not be possible to constrain all of the paths that may lead to a system

failure. These execution paths, referred to as traces, are able to be searched by the supporting software, employing the sequence of events, referred to as the model topology. Both the topology and the semantics formulate the necessary aspects needed to systematically query the model and identify patterns attained by the model.

Key points on patterns:

- (a) Identifying patterns in the model links the developer's interpretation of the model to an exhaustive software query of the all possible outcomes, exposing the underlying construct of the architecture. Using a query ensures that every pattern is found, even among thousands of potential results.
- (b) These patterns were derived interactively from the behavior description, not superimposed on the model of the system. These patterns may encompass all decision points of the model, or only a segment of the decision model.
- (c) After the core patterns are defined, the user may elect to relax the simplification constraints, while keeping the logical, design, and definition constraints. Keeping these patterns in place assists the developer to sift through the additional traces now enabled again by the model.

6. *Analyze the model.* After execution, two forms of analysis were conducted on the model results: (a) identifying the likelihood of occurrence of each particular trace, and (b) deriving an N-squared diagram or Design Structure Matrix (DSM) [10]–[12] of the results, so that all possible interactions can be shown in a single view-point.

- (a) *Probability:* The probability of each sequence of events was derived by developing a post-execution script that employs Bayes theorem of conditional probability to the decision points of the behavior analysis. The core capability is now executable directly within MP-Firebird, version 3.5 and higher. Examples of Bayes theorem included those outlined by Duda [13], and were employed for verification of the approach. Significantly, the conditional probability construct needs to be fully consistent with the constraints employed by the model. The constraints fundamentally change the belief-network of the probability structure. Further, the constraints are a form of conditional probability, often written as an if-then statement. As an example, consider a constraint that “if a certain condition occurs, the trace is not valid and so the subsequent events have a probability of zero.”

- (b) *The DSM:* Since each trace is represented as a directed graph, without loops, deriving a

DSM is straightforward. The directed graph consists of nodes, comprised of the events, and edges, comprised of all of the interactions among the events. Each edge includes a source node (source event) and target node (target event). By tabulating each interaction in a matrix form for each trace, and then building this tabulation from trace to trace, the developer can ascertain a compilation of all interactions within the schema in one view. A script was developed in order to automatically derive the DSM from an MP output file. Simple inspection of the DSM may include consideration of the rarely employed interactions were expected to exist, and if problematic consider some form of design constraint to eliminate a potentially unwanted behavior. More complex analysis may include a means to form groupings of events that have a high degree of interaction. Employing a form of a clustering algorithm, developed by Thebeau [14], this algorithm was applied to the DSM derived from MP, providing a means to systematically gain additional perspective of the interactions derived from the system architecture.

Example Problems: The dissertation details several example problems that demonstrate the methodology and inform the findings of the dissertation. Three example problems were extensively investigated, including a failure mode problem that examines the interactions among a leader and subordinate(s), an SoS data exchange model that examines the hand-off of critical information to complete a mission, and a designed SoS model that includes layers of design from the component of participant systems to the combined system capability. These examples demonstrate the depth of analysis associated with the proposed methodology, illustrating the level of detail that can be accomplished even using models at a higher level of abstraction. These models also serve as a means of verification of the methodology.

Three additional models were also demonstrated, but described in a more general sense, including traffic control, swarm behavior, and a business process example. These additional models demonstrate the application of the methodology across multiple domains of research.

Findings: Researching patterns within behavior model architectures facilitated several findings, including the following:

- The probability of occurrence of any particular sequence of events was demonstrated using a Bayes model [13] for conditional probability. The developer assigns conditional probability for each given event that must be consistent with the constraints of the model.
- The N-squared diagram and the associated DSM are readily produced by executing the model, and inspection of this diagram provides a means to identify all possible interactions of the

model within one view-point.

- The DSM facilitates extended analysis including clustering algorithms that identify regions of close interaction among events, supporting a means to incorporate modular segments within the system design.
- Applications include the use of a behavior model the methodology construct to evaluate emergent properties of a System of Systems (SoS), with dependence upon capturing key interactions among multiple systems.

Key points on the dissertation contribution:

The main contribution of this dissertation is a methodology that enables the developer to describe the behaviors of a particular system, define the interaction of the system with other systems, users, and the environment, add constraints to the model, derive patterns from the set of possible outcomes, analyze and interpret the results, and re-use the model for other problems of interest. Subsequent attributes of this contribution include the following:

- A taxonomy for the model constraints that establishes a formal definition of system requirements, derived interactively with the model,
- Identification of factors essential to the construct of the behavior model, particularly related to gaining SME input throughout the steps of development, validating and verifying the model through iterative process of execution, aligning the analysis to be consistent with model constraints, and employing consistent semantics within a defined ontology,
- Example problems illustrate the methodology,
- Instructions on how the reader may employ the methodology, and
- The method, schemata, scripts, and applications as listed within the body of the dissertation, making them available for follow-on research.

References

- [1] B. P. Zeigler, “A note on promoting positive emergence and managing negative emergence in systems of systems,” *The Journal of Defense Modeling and Simulation*, vol. 13, no. 1, pp. 133–136, 2016. [Online]. Available: <https://doi.org/10.1177/1548512915620580>
- [2] L. B. Rainey and M. Jamshidi, *Engineering Emergence: A Modeling and Simulation Approach*. Boca Raton, FL: CRC Press, 2018.
- [3] F. P. Brooks, *The Mythical Man-Month*. Boston, MA: Addison-Wesley, 2010.
- [4] Monterey Phoenix Home Page. (2018). Naval Postgraduate School. [Online]. Available: <https://wiki.nps.edu/display/MP/Monterey+Phoenix+Home>
- [5] M. Auguston, “Software architecture built from behavior models,” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 5, pp. 1–15, 2009. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/1598732.1598733>
- [6] M. Auguston, “Monterey Phoenix, or how to make software architecture executable,” in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pp. 1031–1040. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/1639950.1640075>
- [7] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, 2012.
- [8] D. Jackson, “Alloy: A lightweight object modelling notation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: <http://doi.acm.org.libproxy.nps.edu/10.1145/505145.505149>
- [9] Defense Acquisition University, “Defense acquisition guidebook,” vol. 19, p. 1248, June 2013. [Online]. Available: <https://acc.dau.mil/CommunityBrowser.aspx?id=511646&lang=en-US>
- [10] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, “A model-based method for organizing tasks in product development,” *Research in Engineering Design*, vol. 6, no. 1, pp. 1–13, 1994. [Online]. Available: <https://link.springer.com/article/10.1007/BF01588087>
- [11] T. R. Browning, “Applying the design structure matrix to system decomposition and integration problems: a review and new directions,” *IEEE Transactions on Engineering management*,

vol. 48, no. 3, pp. 292–306, 2001. [Online]. Available: <https://ieeexplore.ieee.org/document/946528>

- [12] S. D. Eppinger and T. R. Browning, *Design Structure Matrix Methods and Applications*. Cambridge, MA: MIT Press, 2012.
- [13] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. New York, NY: Wiley Interscience, 2001.
- [14] R. E. Thebeau, “Knowledge management of system interfaces and interactions from product development processes,” Master’s thesis, Dept. of Eng. and Mgt., MIT, Cambridge, MA, 2001. [Online]. Available: <https://dspace.mit.edu/handle/1721.1/29168>

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I am sincerely grateful to so many people who have made this dissertation possible. As many doctorate candidates have experienced, the published work represents only about one-fourth of the total effort put forth in the research. My experience affirms that trend.

The dissertation supervisor and chair, Dr. Kristin Giammarco, provided tremendous support throughout the process. Her patience, professionalism, and expertise has helped me to think through the next logical steps in the research, and often just serving as a sounding board was so helpful.

Dr. Mikhail Auguston has been so generous with his time and willingness to share his background from the computer science domain. I think that greater advances occur when we gain significant contributions from other domains of research and apply these contributions to new problems. He has caused me to think and communicate in new ways, outside of my realm of systems engineering.

The Systems Engineering Department head, Dr. Ron Giachetti, provided extensive review of the research. I am amazed at how quickly and thoroughly comments were provided on hundreds of pages of content.

Distinguished Professor Dr. Clifford Whitcomb provided expertise across research domains, having been familiar with concepts of behavior analysis.

Dr. Raymond Madachy also provided consistent advice, ensuring that the dissertation kept on track.

Also from NPS, Dr. Paul Montgomery and Professor Ron Carlson encouraged me to get started in this endeavor. Their engagement over the years was both insightful and necessary to keep my focus on the right next steps.

Also at NPS, my fellow doctorates and candidates Dr. Katy Giles, Gary Parker, Chris Wolfgeher, Dr. Bonnie Johnson, and Ken Hollinger, thank you so much for the camaraderie and continued support.

Ms. Kelly McCool, CAPT Al Mousseau, Mr. Spencer Crispell, Mr. Andrew Lieux, Mr. Wayne Wilhite, Mr. Jaime Guerrero, and many others from the Naval Air Systems Command and Naval Air Warfare Centers ensured that I kept on track over the extended period of time that was required to complete the research.

To my dearest, Katherine, who endured many hours of *our* time being used for research. These hours were quite isolating, and you have always encouraged me though them.

Our children and their spouses have always been a great encouragement; thank you Jake, James and Hannah, and Elyse and Matthew. Thank you also to my siblings and their spouses, Maryann, Reg and Jim, and Jim and Helen. I need all of you.

May all things be subject to the will of our heavenly Father, through the completed work of Christ our Lord, and by the empowering of the Spirit. Soli Deo Gloria.

CHAPTER 1:

Introduction

In this dissertation, the author defines a methodology to derive inherent behavior patterns in system architectures; these patterns consist of an observable sequence of events that manifests in different combinations across many domains. A developer of a designed system of systems will naturally emphasize the intended paths of execution, but the sequences of alternate paths introduces the unexpected patterns, leading to a particular outcome of the behavior that may be either positive or negative. These patterns become more interesting when the developer considers not only the desired sequence, but also the possibility of alternate undesirable paths of execution of the sequence. These alternate paths may produce either positive or negative outcomes from the viewpoint of the developer. Deriving these patterns of system behavior early in the development cycle enables the user to characterize large sets of potential outcomes of a system, find similarities across multiple domains, and re-use successful architectures. In order to ensure that the system under development accomplishes its intended function and that the system does not exhibit unwanted functions, the developer needs to evaluate multiple paths of execution of the model: (1) those that are expected, or designed-in to the system and (2) those that are also inherent in the system, but not expected by the developer. The system behavior may exhibit both positive and negative patterns [1] that introduce design objectives to be managed by the developer.

The possible sequences of events of a behavior model could include many thousands of different outcomes, based on the level of complexity of the model and the scope of execution needed to derive the expected behaviors. Expanding upon this thought, if a sequence of events includes ten consecutive decisions with two alternatives, this results in over a thousand ($2^{10} = 1024$) possible outcomes. A reasonably complex system may include thirty decision points, resulting in over a billion outcomes, as indicated in Table 1.1. The developer may be expected to manually inspect on the order of tens or perhaps hundreds of sequences, and so an automated means to inspect behavior model becomes vital. Analysis needs to provide a means to to “detect, classify, predict, control, and visualize behavior in system of systems” [2]. This dissertation outlines a methodology to manage large numbers of outcomes, demonstrating comprehensive analysis a model with tens of millions of potential results, bounded only by the practical computational limits available to the developer.

Benefits of identifying patterns in system-of-systems architectures include the following:

- logically robust system architecture, developed interactively with the model that exposes

Table 1.1. Number of alternative paths, given two options at each decision point.

Number of decisions, n	Number of alternatives, given two options, 2^n	
1	2	singles
2	4	
3	8	
4	16	tens
5	32	
6	64	
7	128	hundreds
8	256	
9	512	
10	1,024	thousands
20	1,048,576	millions
30	1,073,741,824	billions

potential flaws,

- identification of both positive and negative behavior patterns inherent in the system architecture,
- re-use of successful models with the potential benefits of reduced cycle-time to develop new models, reduced labor cost of development, [3], and increased likelihood of developing a successful model.

Segments of this research have been published in open literature, and portions of these publications are repeated within this dissertation, used with permission. The following lists the publication references, [4]–[6]:

J. Quartuccio, K. Giammarco, and M. Auguston, “Identifying decision patterns using Monterey Phoenix,” in *12th International Conference on System of Systems Engineering (SoSE)*, 2017. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/7994952>.

J. Quartuccio, K. Giammarco, and M. Auguston, “Deriving probabilities from behavior models defined in Monterey Phoenix,” in *12th International Conference on System of Systems Engineering (SoSE)*, 2017.

J. Quartuccio and K. Giammarco, “A model-based approach to investigate emergent behaviors in systems of systems,” in *Engineering Emergence: A Modeling and Simulation Approach*, L. Rainey and M. Jamshidi, Eds. CRC Press, 2018.

1.1 Background

Understanding the behavior of systems of systems has become increasingly relevant. Model-Based Systems Engineering (MBSE) enables early definition of a system, delineating capability objectives, then deriving the system requirements, functions, synthesis, and leading to system verification. As systems of systems are developed using the tools of MBSE, rigorous analysis of all behaviors of the operational system(s) must be well-understood and constrained to align with the intended design.

Advances in analysis methods and rigorous evaluation of computer and software systems enable advances in capability that can be applied to systems engineering of physical systems. This research seeks to exploit this opportunity by investigating ways to describe the behaviors systems of systems. This research also proposes a behavior-based methodology to identify emergent properties of engineered systems of systems. Logic analysis, modeling, and evaluation of example cases provide verification of the capabilities and identify potential advances of MBSE tools.

Understanding all possible behaviors of a system of systems is the responsibility of the systems engineer. The implementation, or lack of implementation of available MBSE tools have impacted successful integration of systems of systems and the effectiveness of delivering advances in capability. The systems engineer or systems integrator holds the responsibility to implement advanced capabilities in a safe, achievable, and predictable manner. Gaps in understanding of behavior early in the development phase leads to increased costs, extended schedule, or failure of the program. Without adequate definition, discovery of problems often occurs late in the development cycle when costs are high. Department of Defense programs have marginal success delivering systems of systems within cost and schedule [7]. The methodology demonstrated within this dissertation helps the developer to resolve ambiguity of the system definition; thereby improving the posture for successful program execution.

1.1.1 Behavior Model Concepts within a System of Systems (SoS)

The following concepts were useful tools for shaping the methodology for detecting, classifying and cataloging behavior patterns:

The concept of an abstraction: An abstraction is disassociated from a particular instance or occurrence. For example, the role of a leader is an abstraction for particular kinds of leaders, such as a surgeon, pilot, professor, or manager. Employing a model at a relatively high level of abstraction enables consideration of the structure of the model without engaging in the level of detail needed for production of the system, such as software coding, physical design, and fabrication.

The concept of an event: Let us consider an event as an abstraction of any particular activity. An event has an observable beginning and ending, with a duration of zero to some positive value of time. To read is an event that requires the participant to engage in the activity over time. The point at which one would lose balance is an instantaneous event, since at one point in time balance is maintained until it is lost.

The concept of relationships among events: There are two fundamental relationships when considering the association of events among other events. *First*, consider the relationship of “inclusion,” such that an event exists within some level of hierarchy. For example, an event of driving a car includes sitting behind the wheel, holding on to the steering wheel, depressing the accelerator, and depressing the brake pedal. Events may also be shared such that a person driving the car and another person walking along the sidewalk may participate the same experience as the car passes the pedestrian. *Second*, consider the relationship of “precedence,” that defines the temporal relationships. Precedence includes the specified ordering as well as the intentional non-specified ordering of events, or concurrency. An example of a precedence relationship is that writing a letter must occur prior to reading that same letter. Conversely, concurrent events may include reading the letter while riding in a train. Many other relationships may exist among events, that may add context to the interaction, but are included as an additional description. Examples include binary relations that describes the interfaces between architecture elements, derived from Vitech CORETM documentation [8], as follows: achieves, achieved by, basis of, based from, builds, built from, contains, contained by, decomposes, decomposed by, documents, documented by, encompasses, encompassed by, exhibits, exhibited by, implements, implemented by, includes, included by, incorporates, incorporated by, refines, refined by, specifies, specified by, triggers, triggered by.

These concepts are formalized within the Monterey Phoenix (MP) Firebird tool [9] which was used extensively in order to facilitate the findings of this dissertation. MP affords a means to build behavior, employing light-weight formal methods, models that are adaptable to systems engineering, and extensible to levels of abstraction suitable for the engineering and design of systems of systems [10], [11]. These structures, coded with a formal language, are then executed to form a comprehensive series of traces, each representing a unique series of events.

MP was inspired by the analysis of software structures, specifically by the Small Scope Hypothesis [12] and the associated Alloy Analyzer environment [13]. Consistent with this construct, MP finds all possible outcomes of a behavior model within the scope of execution. The Small Scope Hypothesis suggests that most problems in the architecture can be found with only a few iterations of the model [12], and so execution of the model affords the developer to be assured that all

permutations have been exhausted, within the scope of execution. The execution environment supports assertion checking, trace visualization, annotation of each instance of behavior, and post-execution results in graphical and data file form. These data furthered the contribution of the dissertation.

1.1.2 Enabling Technologies for Designed Systems of Systems

Advances in digital computing through exponential decrease in the size of integrated circuits [14] has enabled an unprecedented level of interaction among systems. For over 50 years, Moore's Law, indicating the number of transistors per square inch doubles every year, has been expanded as a general rule for integrated circuits, even as computing architecture reach the atomic level. Advances in digital sensor technologies, data interfaces among systems, and high speed processing have made interactive system of systems possible.

- *Sensor systems:* Electro-Optical/Infra-Red (EO/IR), Light Detection and Ranging (LiDAR), Radio Frequency (RF), and Global Position System (GPS) have been implemented in a wide range of sensors across many platforms including autonomous vehicles.
- *Data links and communication:* Networked sensor systems via RF data links and other communication forms enable advances in localization and simultaneous fusion of multiple physical locations.
- *Computing power:* Enabling behaviors with real-time algorithms running on distributed multiple systems.

1.1.3 Potential Sources of Unfavorable Behavior

Unwanted behaviors may be introduced to the system of systems through a variety of sources; however, an emphasis is often on the communication and data transfer between constituent systems. Reliance of a communication link, user interaction, and adverse environments introduces many possible sources of negative behavior.

- *Information latency:* Operating on latent data may be induced by filtering, physical responses, communication protocol, external sources, or environmental conditions. Examples include the following:
 - unexpected data latency induced by filtering, data collisions, or other source,
 - overload of networks, and
 - unplanned or unpredictable data publishing schedule.

- *Hardware response:* Physical limitations or responses of the physical system unanticipated by the developer, nonlinear failure, progressive structural changes due to cyclical loading to the structure, or state changes of the any component or transport utility (e.g., liquid to gas of a cooling function) may cause unanticipated response of the system. Examples include the following:
 - change of material state,
 - unexpected hardware response, and
 - hardware failures.

- *Environmental conditions:* Adverse conditions include off-nominal conditions of heat, cold, power availability, atmospheric changes that impact critical phenomena of the system. Examples include the following:
 - unexpected environmental conditions including electromagnetic interference, heat, humidity, sand, and dust,
 - obscured sensor input, and
 - unintended feedback.

- *Communication delay or failure:* network construct, interference in radio frequencies, saturation, and network management, network contention, or other imposed protocol. Examples include the following:
 - dependence on data transfer, yet operating out of phase or with partial data,
 - ineffective network management, and
 - unexpected breaks in communications.

- *Human user:* induced failures due to poor design may impact timing, response, and throughput; thereby introducing errors. Examples include the following:
 - user induced failures by either operators or maintainers, and
 - malicious actors.

- *Design constraints:* limitations to the system behavior may inhibit both favorable and unfavorable aspects of the system and become critical factors within a System of Systems (SoS) design. Examples include the following:

- unintended effects of constraints, including limits, Boolean logic, or weighting factors, and
- design for average responses, not taking into account the normal range of hardware response, and outlying response.

1.2 Statement of the Problem

The dissertation problem statement is as follows:

Behavior analysis of system of systems architectures can produce hundreds or thousands of possible traces, each representing an individual thread within a use case, such that comprehensive analysis becomes difficult, if not intractable, for the developer to effectively and completely evaluate.

The developer needs to evaluate multiple paths of execution of the model – those that are expected, or designed-in to the system as well as those that are not expected by the developer. The results could include thousands of possible outcomes, based on the level of complexity of the model and the scope of execution needed to derive the expected behaviors.

The developer that needs to deal with a large set of execution paths of a behavior model may benefit by gaining some methodology and automated means to determine whether repeatable patterns of behavior exist within those execution paths. Potential benefits of identifying patterns in System of Systems (SoS) architectures include the following:

- Derivation of a logically robust system architecture, developed interactively with the model that exposes potential flaws,
- Identification of both positive and negative patterns of behavior inherent in the system architecture,
- Re-use of successful models with the potential benefits of reduced cycle-time to develop new models, resulting in the following:
 - Reduced labor cost of development [3]
 - Increased likelihood of developing a successful model,
- Provision of a structured means to examine and characterize large sets of possible system behaviors.

1.3 Research Objective

The objective of the research is to develop a methodology and automated support to analyze architectures for desirable and undesirable behaviors arising from system of systems interactions.

This objective is to be pursued by defining architectures within the Monterey Phoenix (MP) construct [15] as a means to capture system of systems behaviors. The resulting traces capture all possible behaviors within a given scope of execution. Supporting efforts include the following;

1. Derive a general methodology to analyze behaviors captured within system of system architectures,
2. Design and use an automated algorithm, within the MP language, to identify patterns in system architectures,
3. Demonstrate that the proposed research can be employed to an array of problems,
4. Demonstrate that the methodology can evaluate large sets of potential outcomes.

1.4 Purpose of the Research

The purpose of the research is to equip practitioners of MBSE with a methodology to investigate inherent behaviors of systems being developed, including logic analysis in combination with modeling and simulation of the system (or SoS) architecture.

The method addresses both the desired and undesired set of responses of the system at each behavior event, producing a large volume of possible outcomes of the systems. Then, by employing constraints on the model behavior, the developer reduces the possible outcomes to a manageable and expected results. These constraints become formally-defined system requirements.

Furthermore, by defining the meaning of each event and employing semantics to the model structure, the methodology interrogates the architectures and derives the associated templates that capture the behavior patterns of the system. These templates form a repository that is available for future use that intrinsically identifies the patterns of behavior that are exhibited in the new model. The developer may then derive the applicable requirements, as constraints, to impose on the new system.

In so doing, the developer is able to refine predictable and consistent behaviors of the system. The

intent of the methodology improves the design approach of a system of systems; thereby furthering the body of knowledge within systems engineering and adds rigor to its practice.

1.5 Theoretical Framework

With an application to software analysis, Jackson developed a means of automating analysis of computer code, including the foundations of logic, language (syntax), and semantics [12], [13], [16]. Jackson proposed the Small-Scope Hypothesis, stating that most problems can be found in relatively few iterations. This theory depends upon a coherent and formal description of the software structures under consideration. The Alloy Analyzer, developed by Jackson implements this construct as a light-weight formal method (<http://alloy.mit.edu/alloy/>). The exploration of behavior models conducted within this dissertation will support or discount the application of the Small-Scope Hypothesis as applied to behavior models in systems engineering.

Concepts formulated within this dissertation rely upon the Small-Scope Hypothesis, applied to systems architecture. Auguston developed the Monterey Phoenix (MP) [10], [11] as a tool that captures behavior models including component level actions and system level interaction among the constituent systems, users, and environment. This construct readily implements systems engineering architectures, executable as MP Firebird (<https://4.firebird.nps.edu/>). MP adds formal definition at a high level of abstraction so that the fundamental underpinnings of a system model can be checked to hold together. As a light-weight formal method, MP employs a complete specification implemented through event grammar that is consistent with fundamental axioms that ensure its accuracy. In lieu of proofs, MP derives all possible sets of alternative events, known as traces within the tool.

The reader may ask why the research relies on evaluating only a small portion of the model results, why not check them all? Certainly, the developer needs to be aware that the verification and validation of the model should be limited to the scope of evaluation of that model. The methodology promoted by the dissertation exhaustively finds all of the output up to the scope of execution. The key point is that at some level of execution, all possible behaviors have been identified, and any projected future behavior is a combination of those that have already been evaluated. The methodology evaluates large data sets, revealing the combinations of templates that enable a complete evaluation of that data set at tractable levels of investigation.

1.6 Scope of Research

The scope of this research centers on the design of systems of systems that are applicable to the Department of Defense (DoD). The pattern-finding construct, however, is anticipated to be applicable to a wide range of patterns found in multiple disciplines, which shall be discussed as future work. Applicable domains worthy of exploration include the following:

- Designed systems of systems
- Human interface with designed systems
- Traffic patterns
- Telecommunication networks
- Financial models
- Business processing
- Logistics management
- Decision networks
- Biological systems behaviors

The fundamental models that were developed include the following:

1. *Decision Model*: Outlines a failure mode problem experienced in a hierarchical workforce, dealing with the perception of a routine procedure that leads to a critical problem. Inspired by a leadership book by Syed [17].
2. *Object-to-Effect Model*: Reflects an operational task that relies upon multiple sensors to produce compiled information that is sufficient for some critical action. Derived from Dahmann et al. [18] and Antul et al. [19].

1.7 Assumptions and Limitations of the Study

The research depends upon defining System of Systems (SoS) behavior analysis such that the associated models are readily predictable or readily reproduced by simulation. System of Systems (SoS) behavior models may exhibit a level of emergence, correlating to “simple and weak” categories, as defined by Maier [20], as listed in Table 2.3, and discussed further in Section 2.1.2.2. By Maier’s definition, “spooky” categories of emergence would not be able to be modeled and then simulated; thus for these problems it is not possible to resolve the associated behaviors using the approaches that have been investigated thus far.

Inherent limitations of the study include finding tractable and reasonable solutions, as limited by

computing power normally available within an educational and university setting. A personal personal computer was used for local and on-line access to MP-Firebird, MATLAB[®], Vitech-CORE[™], Innoslate[®], and Microsoft[™] tools were used for the analysis.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Prior Work

2.1 Review of Literature

This section provides a literature review, reflecting the current body of knowledge of systems engineering modeling relevant to the development of systems of systems and understanding the observable behaviors of these systems.

The intent of this section is to provide an integrative study of previous work in order to form a basis from which the author has forged a new methodology that enables the systems engineer to identify behavior patterns by developing, analyzing, and executing system architectures, as outlined in the following chapter.

Using this methodology, the systems engineer may then be positioned to re-use not only the specific pattern, but also re-apply key concepts and restrictions gleaned from each new model that applies the new methodology. Aspects of behavior of interest include both the identification of positive processes, to capture what the system needs to accomplish, as well as the identification of negative processes, to capture the necessary behavior restrictions.

The following outline of the literature overview provides the greater context of how the dissertation research, is positioned in the greater literature. A brief statement of relevance of previous work is listed here, while more detail is added in each of the following sections of this chapter.

- ***Systems science*** investigates the nature of systems, seeking to build upon our knowledge of systems in a way that is testable and verifiable.

Relevance of systems science to the dissertation methodology: To the extent that similar processes occur across domains of research enables traditional findings in one discipline to be applied to another discipline in a novel and unique way. Types of activities associated with system processes were projected in this section. These were then used to form the basis of identifying behavior patterns as part of the dissertation methodology.

- ***Systems engineering*** establishes a discipline by which developers elicit requirements, allocate functions, and synthesize a final product. Ongoing research includes ways to apply

traditional methods and the development of novel methods to address systems of systems, complex adaptive systems, and managing the effects of emergence where the resulting behaviors are not derived by reducing the elements of the system, but are exhibited because of the interaction of the systems with external elements.

Relevance of systems of systems to the dissertation methodology: Since the behaviors associated with systems of systems are dependent not only upon the functions within the system, but also upon the interactions among multiple systems, *behavior modeling* must be able to capture both the actions of each individual system and the interactions among the systems.

- ***Modeling and simulation*** enables the developer with a means to represent the actual system to a simplified representation of the system. A representation of the system may be *conceptual*, such as a pilot's perception of how an aircraft will respond to manipulation of the controls, *physical*, as in a wind tunnel model of the aircraft, *computational*, such as a physics-based model of aircraft dynamics.

Relevance to the dissertation methodology: Computational models were developed and employed to capture the behavior architecture of the systems under consideration throughout the execution of this research. This segment of literature search provides an overview of the modeling and simulation in order to provide context to the greater body of work. Understanding the behavior of the system under development using behavior modeling ought to occur prior to investing a significant amount of resources in detailed design that includes physics-based, agent-based, or discrete event models. The author proposed a taxonomy of modeling and simulation, recommending behavior analysis to be accomplished prior to investment in detailed analysis.

- ***Model-Based Systems Engineering (MBSE)*** has enabled significant advances to systems engineering practice, particularly through the advancements in Systems Modeling Language (SysML) and the associated profiles that enable particular viewpoints to be produced. Projected advancements to MBSE includes (1) integration with simulation and analysis, (2) defined theory, ontology, and formalism, and (3) repositories crossing multiple domains [21].

Relevance of MBSE to the dissertation methodology: The methodology to determine behavior patterns in systems engineering architectures provides a means to meet the requirements

for a defined ontology, formalism, and repositories that cross multiple domains. As MBSE evolves in order to meet future challenges associated with systems of systems, behavior analysis needs to be a significant part of early model development so that the employment of future systems have well-understood capabilities and limitations.

- **Ontology** describes the relationships among elements of the systems model. Historically within the DoD, systems models in support of MBSE have been kept consistent by the developers within each system under development, with very little consistency across the programs. Glossaries developed under the authority of the joint chiefs and the service compartments include the DOD Dictionary of Military and Associated Terms (JP-1), the Universal Joint Task List (UJTL), and the Universal Naval Task List (UNJTL), Air Force Universal Task List (AFUTL), and the Joint Mission Essential Task List (JMETL). Since these glossaries, however, often have multiple interpretations, and the usage of the terms are often context-dependent. Efforts have been made to catalogue these glossaries in model form, within SysML, however the relationships of the terms remains largely undocumented in a modeling language form.

Relevance of ontology to the dissertation methodology: The concepts that enable reuse of model components relies upon a comprehensive repository. In order to be effective, this repository relies upon the consistent semantics across the domain of interest. Automated reuse of the behavior models, following the methodology as outlined in this dissertation, will be possible to the extent employed by a governed ontology.

- **Formal methods** enables a means to capture the content of the model elements in such a way that enables mathematical and logical composability. In this way, reasoning may be deduced from the model.

Relevance of formal methods to the dissertation methodology: This dissertation employs MP analyzer as a means to describe system behaviors. Prior to this dissertation, there had not been a means developed to identify patterns in system behaviors within MP. The contribution uses the inherent construct, including the language, grammar, and query functions in order to sort through the complete set of outcomes derived from the model. The approach uses behavior constraints in order to refine the response of the developed system. The dissertation method intrinsically derives these patterns as opposed to superimposing a fixed construct.

- *Systems engineering resources* provided as a reference.

Added details include the application of Model-Based Systems Engineering (MBSE), ontologies of these models, and Modeling and Simulation (M&S) to address SoSs. Finally, applying formal methods to pattern identification of system architectures finalizes the context related to the area of contribution of this dissertation.

2.1.1 Systems Science

Systems science investigates the nature of systems, seeking to build upon our knowledge of systems in a way that is testable and verifiable. Applying scientific disciplines to the study of systems enables rational predictions of observable behaviors of the system. A general systems theory posits that unifying concepts apply across domains of research. These domains originally applied to the nature of sciences of biology, psychology, and social science. The concepts have also been extended to include designed system processes, such that processes of the same form exist both in natural and engineered systems, including communication, feedback, transport, storage, and cycles.

2.1.1.1 General systems theory

General Systems Theory (GST), as developed by von Bertalanffy [22], [23] postulates that a common set of rules governs processes across many naturally occurring systems, including those from the scientific domains of physics, chemistry, and mathematics. These concepts have been expanded such that GST applies to economics, social sciences, and psychology. More recent research has applied these concepts to systems engineering [24]–[29]. For those instances where the theory holds true, a large investment in the research in one domain would also apply to the another applicable domain; thus, the concept of re-application of knowledge would extend to enable breakthroughs in multiple fields. Several aspects detract from the application of GST concepts including

- domain-specific language and nomenclature,
- isolated communities of interest, and
- lack of governance or unifying body.

von Bertalanffy, an Austrian biologist, is considered as the “grandfather” of general systems theory [22], [23], contributed to the advancement of general systems theory, primarily through observations of natural science, but also advanced the concepts that unify thought across multiple disciplines.

- Introduction of the concepts of isomorphism such that the same form of behavior occurs across multiple disciplines [30].
- Introduction of the concept that in addition to the reduction of an organism as that sum of its parts, and also “a complex of elements in mutual interaction ” [30], [31].
- Introduction of the concept that a “structure is a slow process, and a process is a very fast structure” [32], [33] form is a very slow function, and a function is a very fast form.

2.1.1.2 Systems process theory and isomorphic systems processes

Systems Processes Theory (SPT) represents a segment of General Systems Theory (GST) as a specific application of general theory, specifically as it applies the concepts of isomorphic processes across scientific disciplines and implemented across application domains.

Fundamental systems processes for a general theory of systems, originally derived by Troncale [24] as 110 processes, and refined by Friendshuh and Troncale [32] to fifty-five processes, as summarized in Tables 2.1 and 2.2. These authors defined a a system process as a “series of steps typical of surviving systems that adequately fulfills a needed system function when considered at the abstract system level” [32]. The authors maintain that the system level process is dependent upon the “’pattern’ of interaction (perhaps we could call it the shape or ’architecture’ of interaction on the abstracted general level) that is the defining element” [32].

Giammarco and Troncale [34] explored systems processes of cycling: oscillations, lifecycles, and recycling and found that in addition to these processes, anti processes were also indicated. An oscillation was identified in the model as “step-foward” followed by “step-backward,” while the anti-pattern was also discovered as “step-backward” followed by “step-forward.” The authors expanded upon the concept of isomorphic processes as follows:

Each domain contains system processes that are described in the language of its home domain. This research builds on the SPT premise that there exist isomorphic system processes (ISPs) [32] that remain constant across many domains, and that these ISPs can be expressed as domain-neutral concepts to reveal insights about how systems work in general. ISPs are termed “isomorphic” because they are found across many, if not all, mature systems and so they are iso- (same) and -morphic (form). (Note that this is quite a distinct use of the term from its use in mathematics). The term “systems” is used because ISPs are hypothesized to be the fundamental mechanisms by which the sustainability of systemness is achieved; that is, they are the minimal steps

Table 2.1. Fundamental systems processes for a general theory of systems, originally derived by Troncale [24] as 110 processes, and refined by Friendshuh and Troncale [32] to these fifty-five processes.

1. Adaptation Processes	29. Limits, Physical & General
2. Allometry, Systems-Level	30. Integration Processes
3. Allopoiesis	31. Metacrescence as a Process
4. Binding Processes	32. Network Structure & Processes
5. Boundary Conditions as a Proc	33. Neutralization Processes
6. Causality Processes (linear vs. non-)	34. Non-Equilibrium Thermodyn-Irrever
7. Chaotic Processes	35. Origins Processes
8. Competitive Processes	36. Output Processes
9. Constraint Fields & Analysis	37. Phases, Stages, Transitions
10. Cycles/Oscilations/Hypercycles as Processes	38. Power Laws, Cross-Disciplinary as a P
11. Decay, Autolytic & Senescent Processes	39. Quantum Processes
12. Development Patterns & Laws	40. Recursive Processes
13. Duality/Complementarity/Counterparity Mech's	41. Redundancy Processes
14. Dysergy as Process	42. Replication Processes
15. Emergence Processes	43. Self-Criticality/Tipping Pts/Catastrophes as Processes
16. Entropy, General (as a process)	44. Self-Organization/Autopoiesis/Autocatalysis
17. Equilibrium & Steady State Proc's	45. Spin Processes
18. Evolutionary Processes	46. Storage Processes
19. Exaptation, Cooption Processes	47. Structure as Process
20. Feedback, General	48. Symmetry, Systems-Level (as a process)
21. Field Processes & Potentials	49. Synergy/Synchrony/Cooperation as a Process
22. Flow Processes	50. Thermodynamic Processes
23. Fractal Structure (as a Process)	51. Variation Processes
24. Functions, System (Purpose)	52. Maximality Principles
25. Growth Patterns & Laws	53. Minimization Principles
26. Hierarchies & Clustering as a Process	54. Amplifiers as a Process
27. Information-Based Processes	55. **placeholder for new processes
28. Input Processes	

for maximal efficacy to achieve system function/purpose no matter what the scale or compositional parts of the system. ISPs are termed “processes” because they perform a similar function/purpose across all these systems by the same “process steps” or developments in an obligate sequence [34].

Considering these systems processes as a collective set, it becomes evident that certain sequences will contain combinations of processes; however, relying on the nature of an event as observable, there may be a means to expand upon the investigation of the oscillations to other types of behaviors.

Table 2.2. System-level processes. Adapted from [32].

1. Boundaries as a SysProcess	8. Fields as SysProcesses
2. Chaotic Processes	9. Flows as SysProcesses
3. Competition vs. Cooperation (Synergy) as a SysProcess	10. Fractal Structure & Processes
4. Cycles/Cycling/Oscillation/Hypercycles as SysProcesses	11. Hierarchy as a SysProcess
5. Duality/Complementarity/Counterparity as SysProcesses	12. Networks & their Dynamics as SysProcesses
6. Emergence/Origin Mechanics as SysProcesses	13. Self-Organization/Autopoiesis/Self-Assembly/Autocatalysis as SysProcesses
7. Feedback as a SysProcess	14. Storage as a SysProcess
	15. Symmetry as a SysProcess
	16. Variation Production as a SysProcess

A proposed set of behaviors is listed as follows:

Systems processes that can be defined by behavior models

- *Activity: **Oscillation** [34]*
SysProcess: 4-Cycles
 step forward -> step backward
 step backward -> step forward

- *Activity: **Communication***
SysProcess: 4-Cycles, 9-Flows, 12-Networks
 transmit -> receive
 transmit -> relay -> receive
 transmit -> interrupt

- *Activity: **Command and Control** [35]–[37]*
SysProcess: 9-Flows, 12-Networks
 observe -> orient -> decide -> act
 monitor -> assess -> plan -> execute

- *Activity: **Fire control***
SysProcess: 9-Flows, 12-Networks
 ready -> aim -> fire
 find -> fix -> track -> target -> engage -> assess

search -> detect -> track -> fire -> assess

- **Activity: Formation**

SysProcess: 5-Duality, 6-Emergence

detect -> attract (if too far) -> align

detect -> deter (if too close) -> avoid

- **Activity: Authentication**

SysProcess: 4-Cycles, 14-Storage

request -> check -> confirm -> approve

request -> check -> not confirm -> not approve

- **Activity: Transportation**

SysProcess: 9-Flows

move -> stop -> move

- **Activity: Repository**

SysProcess: 14-Storage

receive -> archive

retrieve -> send

- **Activity: Feedback**

SysProcess: 4-Cycles, 7-Feedback

measure -> control -> measure

- **Activity: Boundary**

SysProcess: 1-Boundary

detect -> deter

contain

2.1.2 Systems Engineering (SE)

The traditional systems engineering discipline has been established as a practical means to develop systems that meet a user's need through a systematic definition of that need, developing an engineering definition of those needs through a system requirements documents (or specification), developing a functional analysis and associated baseline, further decomposing sub-segments or subsystems through well-defined individual components, establishing a physical configuration, and then verifying the component, subsystem, and system performance, culminating in final validation of meeting the user need [38]. As such, traditional systems engineering relies upon decomposing the overall system capability to an ever-decreasing level, and then developing a physical solution (including hardware and software) that meets the function of the intended application. In a real sense, the goal of traditional systems engineering is to enable definition from the system to the component level, to fully understand the component behavior, with the assumption that when the components are defined and understood that the final system behavior will be well understood. In this way, the system is decomposable, and the goal is to have all behaviors defined through this decomposition and verified by analysis and test.

Traditional systems engineering processes have been developed in order to develop a solution that meets a defined user need. Wells and Sage [39] define attributes of a system to include well defined boundaries, a hierarchical structure, fully integrated set of subordinate systems (comprised of subsystems, components, and parts), characteristic properties at the boundaries or interfaces, human interfaces within the system and external to the system, and a capability of interacting with the surrounding environment.

The Defense Acquisition Guide (DAG) [40] outlined the classical figure of the systems engineering process to include requirement development, functional allocation, and synthesis, as illustrated in Figure 2.1. This process does follow reductionist principles, and its logic has been effective in the development of nearly all systems that are currently operational.

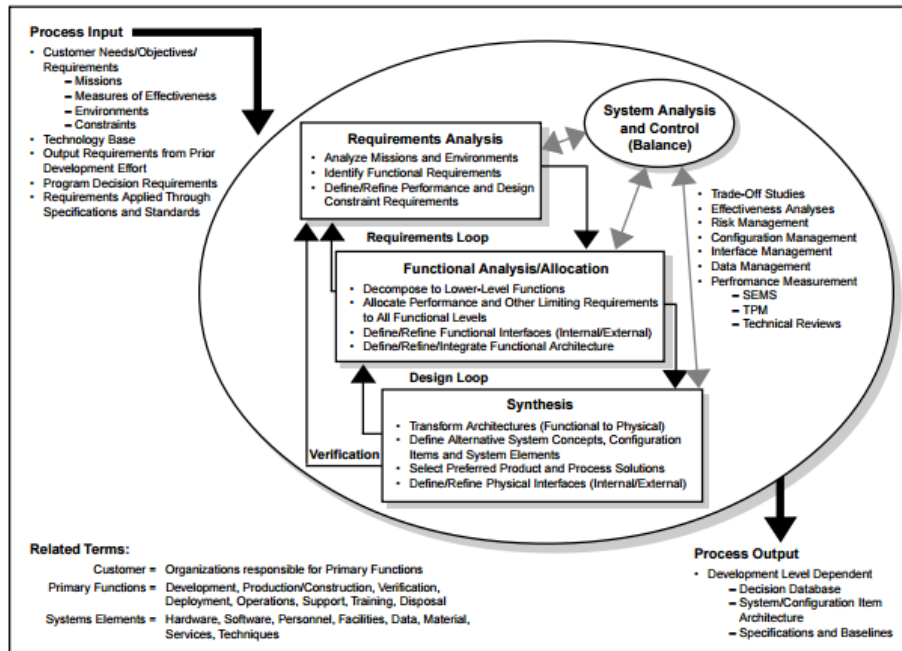


Figure 2.1. Systems engineering process, as described by the Defense Acquisition Guide (DAG). Source: [40]

2.1.2.1 Systems of systems, complex adaptive systems, and emergence

In contrast to traditional systems, systems of systems demonstrate complex behaviors, including emergent behaviors that cannot be deduced by the sum of the individual components. As discussed by Calvano [41], a complex system is characterized by exhibiting tightly coupled interactions of elements within the system, cause and effect of design decisions not being obvious, small causes resulting in large effects, risks including unforeseen emergent properties, influences on decisions being difficult to bound and affirm. [41]–[45]

Jackson et al. [46] defined a SoS as “a coalition of independently operated and managed systems that function together as one unit to efficiently and effectively meet a specific need.” The authors related the importance of key performance metrics as impacted by the autonomy of individual member systems. They offer the use of a nonlinear objective function to determining the optimal configuration while incorporating relevant aspects of the constituent systems.

Keating [47] contended that the expanse of information, interdependence between systems, demand for accelerated deployment, and the holistic nature of systems of systems have accelerated the need for new approaches to deal with the resulting system-of-systems. Complex systems include strong interdependence of agents, emergent collective behavior, and non-decomposable construct

such that the system of systems is greater than and different from the sum of its parts. Keating defined attributes of systems of systems, proposes three perspectives of emergence (philosophical, methodological, and axiomatic), and outlines design and operational considerations for systems of systems. With respect to the design considerations, Keating outlines the importance of robustness to changes in requirements, communication within and external to the SoS, and continuous adjustment to emergence. With respect to the operational considerations, Keating outlined the importance of a continuous scan of emergent trends and patterns, maintaining a strong SoS identity, and developing compatibility of support structures to deal with emergence. Perhaps the most significant challenge for systems engineers is the transition from a process control (often step-by-step) of traditional systems engineering to a methodology control (understanding the concepts and applicability) that is required to develop systems of systems.

Rhodes et al. [48] contributed to the concepts associated with large scale complex SoSs with respect to socio-technical considerations. The authors seek to focus “on understanding how to architect, integrate, manage and transform large-scale enterprises, taking into consideration their overall environment or context.” The authors introduced the concept of an “Epoch-Era Analysis” that develops attributes of performance over time in a series of epochs. The concept may serve to encourage realistic goals that are attainable within cost and schedule.

DiMario et al. [49] addressed the confluence of a system of systems as influenced by a “collective mechanism.” The authors discussed autonomous behavior, collaboration, stressors, autonomy, decision dichotomy, and the resulting emergent system behavior with the lack of a central authority. [49]. The authors also outlined the SoS interoperability spectrum as the intersecting aspects of autonomy, belonging, connectivity, diversity, and emergence as compared to the collective behaviors of coordination, cooperation, and collaboration. [49] Further, the authors also related the analysis with a battle management system, associating the SoS characteristics with levels of decentralized command and control. Under investigation is whether combat goals can be achieved without directed command and control.

Volkert et al. [50] identified the concept of a SoS Performance Measure (SPM) as a means to indicate SoS probability of success, analogous to the Technical Performance Measure (TPM) of a system but applied to a SoS incorporating inherent levels of uncertainty. The authors proposed a means to calculate SoS mission performance based on capabilities of the constituent systems, in terms of system performance and sustainability.

The dissertation research seeks to advance MBSE analysis methodology by incorporating rigorous

architecture analysis prior to systems simulation. Rigorous architecture analysis provides a means to uncover unexpected behaviors within the model. Applying a methodology to potentially derive emergent behaviors found within the model provides a strong motivation for pursuing this research.

Understanding potentially complex behaviors of designed systems of systems motivate the extension of architecture analysis and model checking techniques as tools that the developer can employ. The concept of emergence is associated with the phenomenon exhibited when individual systems interact in a way that is not discernible by decomposing an individual system [51], [52]. While individual system behaviors build to create system of system interactions, these effects may be demonstrated by systems simulation [53]. The outcomes may be positive (beneficial) or negative (detrimental) to the expected system of systems design [1]. Maier [20] classified types of emergence as simple, weak, strong, and “spooky,” as outlined in Table 2.3.

Table 2.3. Classification of emergence, applied to a system of systems. Adapted from [20].

Classification	Description
Simple	Behaviors readily predicted by simple models of individual systems.
Weak	Behaviors readily and consistently produced by simulation, but not by simple models.
Strong	Behaviors consistent with known properties of the individual constituent systems, but inconsistently reproducible results by simulation, not characterized by simple models.
“Spooky”	Behaviors inconsistent with known properties of the individual constituent systems that make up the system of systems.

2.1.2.2 Identification of emergence in systems of systems

Gore [54], [55] outlined an approach using hypothesis testing to identify emergent properties. An applicable example includes responses associated with epidemic models including stocks and flows, based in differential equations. O.T. Holland [53], [56], [57] furthered a formal definition of emergence and developed a taxonomy, formalism definition, modeling and simulation, and topology of the system of systems response. The principle structure proposed by Holland included the modeling and simulation of individual systems, then leading to systems level responses that would not be discernible by investigating these constituent systems in isolation.

Reynolds [58] developed what is credited as the first computational model of swarm dynamics, developing simple rules for individual agent behavior, and then using time-dependent system dynamics numerical solution. Basic tenants of the animal behavior is to move in the same direction, remain close, but avoid collision as neighboring animals. This model kept all agents in contact with each-other, and implemented a nonlinear attraction / rejection formulation.

Developers of System of Systems (SoS) have employed a range of simulation methods as a means to identify and characterize emergent properties. These means have often employed agent-based models or hybrid models that incorporate both agent-based parameters along with a time-space simulation of a dynamic model. Mobile robotics have this type of hybrid response, such that the interaction of systems is dependent upon the spatial relationship of multiple agents. The decision logic employed within these analyses often include a Markov chain type of logic representation. A simple example may be described as may be described as a system may be in either state A or state B, and the alternatives are whether to remain in the existing state or to move to the alternate state. As illustrated in Figure 2.2, the researcher assigns probabilities to these alternatives.

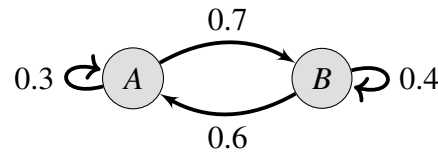


Figure 2.2. Example of a simple Markov chain, consisting of states *A* and *B*, where the probability of moving from state *A* to state *B* is 0.7, while the probability of remaining in state *A* is 0.3. Conversely, the probability of moving from state *B* to state *A* is 0.6, while the probability of remaining in state *B* is 0.4.

Examples of these analyses include the following:

Pentland [59] represented human decision behaviors employing a series of Markov chain models, such that the overall behavior represented as multiple model dynamics, with these decision points broken down as a series of prototypical steps.

Kanert [60] developed an agent-based simulation, but employed graphical means of analysis as a means to identify norms of behaviors. By establishing norms, an instance of non-compliance also relates to non-normative behavior, thus, posing a means for identifying negative emergence. The application is intended for real-time evaluation of cyber-security.

These examples illustrate methods employed for analysis of emergent behaviors within systems of systems. Each of these examples relied upon some architecture describing the interactions and behaviors of the systems. The remainder of this chapter outlines a methodology of how behavior analysis using Monterey Phoenix (MP) can be used to derive the underlying architectures of these types of systems. This methodology emphasizes the behaviors at a high level of abstraction such that subsequent modeling using agent-based, discrete-time based, system dynamics, or hybrid approaches will proceed employing a sound underlying architecture. Specifically, behavior modeling identifies constraints that are necessary to implement the designed system. Analysis of the model also provides insight to the probability of obtaining desired results of the system.

Bouwens [61] applied Systems Geometry (SG) methodology to the problem of identifying emergent behaviors in systems of systems. Bouwens outlined processes, methods, and tools for SG analysis; reliance on the sequence of activities as related to several case studies gave insight to emergent system behaviors, especially as applied to Test and Evaluation (T&E) of SoS. Distributed SoS events yielded lessons learned and observations within the domains of “interoperability, ...constituent system maturity, ...collaboration, ...integration requirements, ...constituent system training, ...resource assessment and utilization, ...analysis and experimentation support, ...and implementing architectural views. [61, 77-80].

Furthermore, Gore [54], [55] outlined an application of hypothesis testing to identify emergent properties. In these studies, examples included physical responses of a sailboat and epidemic models.

2.1.3 Modeling and Simulation

What may be considered as a distinct aspect of Model-Based Systems Engineering (MBSE), Modeling and Simulation (M&S) centers on the analysis of some real system with a model, thereby developing a representation of the system and then simplifying the analysis of that system. A model may take the form of a mental concept of the system, a physical representation, or a mathematical algorithm. This dissertation considers only the employment of a mathematical representation of the system as a computer model. Through execution of the model, a range of observable results is produced, lending insight to the functions of that system.

Consider the postulation that “all models are wrong, ...some are useful” as credited to George E. P. Box, listed as follows:

“Since all models are wrong the scientist cannot obtain a “correct” one by exces-

sive elaboration. On the contrary following William of Occam he should seek an economical description of natural phenomena. Just as the ability to devise simple but evocative models is the signature of the great scientist so over-elaboration and over-parameterization is often the mark of mediocrity.” [62, p. 792]

In applying mathematics to subjects such as physics or statistics we make tentative assumptions about the real world which we know are false but which we believe may be useful nonetheless. The physicist knows that particles have mass and yet certain results, approximating what really happens, may be derived from the assumption that they do not. Equally, the statistician knows, for example, that in nature there never was a normal distribution, there never was a straight line, yet with normal and linear assumptions, known to be false, he can often derive results which match, to a useful approximation, those found in the real world [62, p. 792].

Consequently, Box urges the researcher to realize that any model is an abstraction of the “reality.” In the case of systems engineering, “reality” consists of the final instantiation of the end item or system of interest. Early in development, the requirements, architectures, and analysis are the only representation of the final system, just as an architectural drawing of a residence is the only representation of the residence [63, p. 278-281]; and an early wind tunnel model of an aircraft [64], may be the only representation of the aircraft outer mold line during the concept development phase. The challenge to the model developer is to find the minimum set of system attributes and parameters necessary to answer the associated research questions, without overly complicating the model.

2.1.3.1 Systems analysis through computer simulation

Systems simulation analysis, typically consists of system dynamics models, discrete event models, agent-based models, and hybrid models. These simulation methods follow the construct of the architecture, but also add functional performance parameters relevant to the system design. These system attributes affect the the overall performance of the resulting system, and impact the resulting collective behavior [53]. As an example, a set of aerial vehicles would be able to achieve desired spatial coordination only if performance parameters meet baseline expectations. Typical parameters may include the following:

- *Performance predictions* of the system given a set of expected parameters and an associated range of performance levels (e.g., thrust to weight ratio, drag, acceleration, power).

- *Size, shape, weight, and power* consumption of subcomponents and the corresponding effects on the functional performance, configuration, or design.
- *Reliability analysis* of the system of systems, system, or component that affects the availability or the probability to achieve the mission objective.
- *Capacity analysis* of communication or networks needed to support the coordination among systems.
- *Dynamics and visualization* of combined effects of system motion, such as six degree of freedom simulation.

2.1.3.2 A taxonomy of computer-based simulation models

Within the construct of computer modeling and simulation of dynamics behaviors and interactions within a system, a proposed taxonomy of models and simulation including behavior models, system dynamics modeling, discrete-event modeling, and agent-based modeling, as summarized in Table 2.4 and discussed in the following outline [64], [65]. The latter three types require some level of parameterization, such as speed, range, timing, mass properties, and band-width in order to project the system dynamics. Behavior modeling produces a logically-sound architecture and therefore it should be conducted prior to more detailed analysis.

1. *Behavior models* are developed at a high level of abstraction, prior to populating parameters to the model. Monterey Phoenix (MP) is a behavior model, employing lightweight formal methods, that develops all possible traces, or use cases, within a given scope of execution [66]. Behavior models focus on the essence of the the interactions within a system and with external interfaces, producing a logically-sound architecture.
2. *Discrete-event models* are used for analysis of transactions such as those encountered in financial institutions, logistics and shipping companies, and other behavioral representations. While temporal properties remain integral to the model, the progression of the simulation relies upon completion of a sequence of events. Credit card transactions, commercial airlines transportation, and commercial cargo and shipping systems are examples of processes that are well suited for discrete event modeling. Many aspects of systems of systems are tied to such discrete events. The Discrete Event System Specification (DEVS) formulation was introduced by Zeigler [67] enabling a mathematical representation of system dynamics based

on event sequences.

3. *System dynamic models* are used to simulate classic time-based problems in engineering. Many of these models simplify the system dynamics to a set of first or second order differential equations, code these as difference equation with a variable or fixed time-step, and solve these equations using a numerical method such as Runge-Kutta or other applicable algorithm [68]. Most engineering disciplines use system dynamic models to study phenomena and their applications to include areas of study within dynamics of structures, particle dynamics, flight dynamics, and computational fluid dynamics, with application to many scientific fields. Interestingly, systems theory as applied to other fields also gain benefit from this method, including economics, sociology, and psychology. MATLAB[®] and Simulink[®] [68] and AnyLogic[®] [65] are commercially available packages that model system dynamics. Higher order languages such as any instantiation of C or Java[™] can also be used to develop these models.
4. *Agent-based Models* are more recently developed than system dynamic models and discrete event models. The agent-based formulation describes the individual parameters or variables within the agent, the expected interactions between agents, and the overall context of the environment including number of dimensions, number of agents, constraints of the problem. AnyLogic[®] is a commercially available platform for agent-based modeling [65].
5. *Hybrid models* combine aspects of two or more types of models.

Table 2.4. Types of modeling and simulation. Adapted from [6].

Type	Applications
Behavior models <i>logic-based</i>	System architectures Analysis of relevant behaviors Similarity across domains
Event-based models <i>sequenced by events</i>	Production and manufacturing Transportation systems Logistics
Agent-based models <i>sequenced by an agent</i>	Organizational behavior Animal behavior Biological systems Crowd dynamics Search patterns
System dynamics models <i>sequenced in time</i> <i>physics-based</i>	Flight dynamics Weapon separation dynamics & trajectory Computational fluid dynamics Dynamic structural loading
Hybrid models	Autonomous platforms Combined effects

Key point:

A behavior model forms the essence of system interactions and produce a logically sound architecture. From this basis, further detailed model description may be employed in order to conduct simulation of event-based, agent-based, system dynamics, or hybrid types of models.

2.1.4 Model-Based Systems Engineering (MBSE)

The objective of a system of systems developer extends the traditional systems engineering approaches of requirements definition, functional analysis, and design synthesis in order to attain objectives not possible by a single system [38], [64], [69]. A traditional approach employs a document-based method to capture the system baseline, that might include the specification, interfaces, functional and non-functional requirements, and verification methods. Much of our practice today, captures only a digital version of the documents, and fails to attain the interrelationships of

the system components, let alone system behaviors.

Model-Based Systems Engineering (MBSE) enables a means to capture the interactions and connectivity among components within a system of systems within a model of the system. As outlined by Delligatti, in order to implement an MBSE approach, the user needs to employ a language, method, and tool [70, pp. 5–9].

MBSE has progressed engineering practice from a traditionally paper-based process of development to include requirements, technical drawings, computational analysis of individual problems, to using various relational data structures in order to capture inter-dependencies of these design elements and communicating the source information through a range of viewpoints. MBSE incorporates frameworks such as Department of Defense Architecture Framework (DoDAF) that have gained utility by describing the system under development from the perspectives of various stakeholders, to include operational, capability, system, and service viewpoints. Modeling languages have been developed, particularly by the Object Modeling Group (OMG) through the development of Systems Modeling Language (SysML) and the associated profiles of Unified Profile for DoDAF and Ministry of Defence Architecture Framework (MoDAF) (UPDM) and Unified Architecture Framework (UAF). The intent of this modeling language is capture the meaning of the detailed elements derived by developing the DoDAF viewpoints.

Buede [64] developed a foundational text on systems engineering, outlining the design problem to include definition of the problem, requirements development, system life cycle, stakeholder preference, modeling techniques, verification, validation, and qualification. From the MBSE perspective, Buede employed SysML and implemented the commercial systems engineering from Vitech-CORE™ and Cameo Systems Modeler™.

Maier and Rechtin [71] developed a foundational text on systems architecture, considering both the art and science of the subject. The authors address the relationship of architectures to systems engineering, expanded domains including manufacturing and social systems, models and modeling, and the profession of a systems architect.

2.1.4.1 Architecture frameworks

MBSE enables a means to capture the interactions and connectivity among components within a system of systems.

2.1.4.1.1 Zachman Framework [72] relates the six basic interrogatives of what, how, when, who, where, and why to six considerations that transition a concept to a realization including iden-

tification, definition, representation, specification, configuration, and instantiation. The resulting six by six matrix that is intended as a means to refine any object from concept to its synthesis.

2.1.4.1.2 The Department of Defense Architecture Framework (DoDAF) [73] construct, two viewpoints are particularly reflective of system behavior and associated interactions: (1) the activity diagram, in the form of a Functional Flow Block Diagram (FFBD), Operational Activity Model (OV-5b), and System Functionality Description (SV-4) and (2) the sequence diagram, in the form of an Operational Event Trace Description (OV-6c), and System Event Trace Description (SV-10c).

2.1.4.1.3 The Ministry of Defence Architecture Framework (MoDAF) [74] is the United Kingdom version of a framework, developed as an extension of DoDAF.

While these frameworks established initial concepts of enabling a consistent means of articulating system descriptions, they do suffer from permitting an incomplete capture of relevant aspects of the system design. The associated viewpoints may remain as a partial description of the system, and there is not a formal means to check for completeness. Ang et al. [75] and Mercer [76] outlined particular weaknesses in DoDAF, citing that the description of the subject system may be compliant with DoDAF, and yet be incomplete, poorly composed, and non-coherent. Since DoDAF was defined on a consensus basis, it is only reasonable to expect that limitations in the proposed viewpoints exist. Fundamentally, the construct of the DoDAF viewpoints allow a level of ambiguity and are thus insufficient to adequately address a complete system definition. Ang and Mercer proposed the implementation of formal methods to comprehensively define the system, identifying “who, what, where, when, why, and how” within an improved construct, consistent with formal methods.

2.1.4.2 Modeling languages

The Object Modeling Group (OMG) [77] definition of Unified Modeling Language (UML) [78] and the associated dialect of SysML [79] provide a means for standard definition of syntax, semantics, and exchange of behavior models and associated diagrams including activity diagrams, sequence diagrams, state charts, and use cases. UML and SysML provide a basic language for system diagrams, much like the specifications for technical drawings. However, these definitions are not intended to *evaluate* the content of the architectures that they represent. With current tools, consistency among the behavior related diagrams (e.g., activity diagram and sequence diagram) is often reliant on the architect to ensure that updates across multiple diagrams. This stems from the modeler employing a practice of a focus on the viewpoint as opposed to a focus on the content of the

model and then allowing that content to promulgate across the various viewpoints and associated diagrams. UML and SysML do not enforce this level of consistency.

2.1.4.2.1 Unified Modeling Language (UML) [78] was initiated as a means to implement software object modeling, but was also used to capture basic concepts within systems engineering.

2.1.4.2.2 Systems Modeling Language (SysML) [79] was initiated as an extension of UML, but has since become independent from it since some updates to UML do not exist in SysML. SysML depicts the following aspects of the system [79], as also illustrated in Figure 2.3:

- Structure or hierarchy
- System behaviors
- System requirements
- Parametric values

Specific to capturing system behaviors, SysML includes the following behavior diagrams:

- Activity diagram
- Sequence diagram
- State Machine diagram
- Use Case diagram

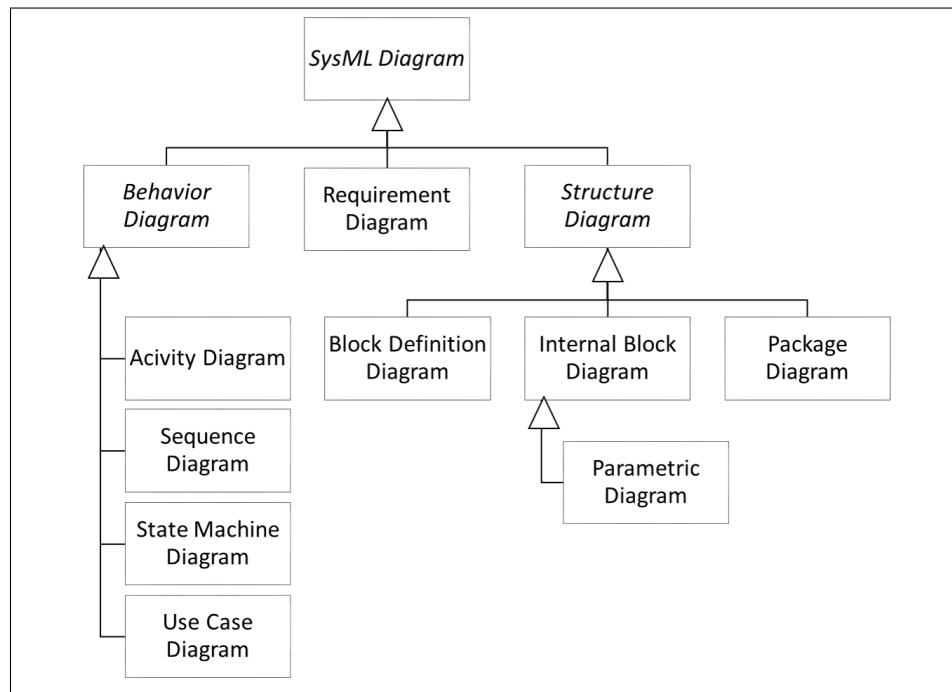


Figure 2.3. SysML taxonomy, where the open arrow indicates a “type of” the indicated block. Adapted from [79, p. 194] and [70, p. 15].

2.1.4.3 Profiles

Consider the following description of a SysML profile from Delligatti:

A **profile** is a kind of package that contains a set of stereotypes. A **stereotype** defines a new kind of model element by adding properties, constraints, or semantics to an existing kind of model element. When you create a profile – a package of stereotypes – you are, in effect, defining a new modeling language that is an extension of an existing modeling language.

You can the **apply** a profile to a package, model, or model library. This means that the package is allowed to contain the new kinds of model elements that exist in the modeling language defined by that profile. [70, p. 195]

The following profiles have been used to support systems engineering programs, with the intent to gain compliance with DoDAF. UPDM and UAF were developed in response to the International Defence Enterprise Architecture Specification Foundation (IDEAS) standard. Hause provided a comprehensive description of the evolution of the standards and profile implementations [80]–[82]

2.1.4.3.1 UML Profile-based Integrated Architecture (UPIA) [83] is a UML profile for systems engineering that has been developed by the IBM corporation. This profile enabled compliance with DoDAF using UML.

2.1.4.3.2 Unified Profile for DoDAF and MoDAF (UPDM) [84] employs SysML in order to derive profiles that are useful to represent viewpoints of MoDAF and DoDAF.

2.1.4.3.3 Unified Architecture Framework (UAF) [85] was initiated as UPDM version 3, designed to capture a combined framework of both MoDAF and DoDAF with updated viewpoints, improved data transfer definitions, and presented in UML or SysML syntax. While UAF adds views to include human interaction, security, and test, the profile also evolves the description and relationships among the viewpoints and incorporates a more readable taxonomy of all of the views. This profile is illustrated in Figure 2.4.

	Taxonomy Tx	Structure Sr	Connectivity Cn	Processes Pr	States St	Interaction Scenarios Is	Information If	Parameters Pm	Constraints Ct	Roadmap Rm	Traceability Tr	
Metadata Md	Metadata Taxonomy Md-Tx	Architecture Viewpoints Md-Sr	Metadata Connectivity Md-Cn	Metadata Processes Md-Pr	-	-	Conceptual Data Model, Logical Data Model, Physical schema, real world results	Environment Pm-En	Metadata Constraints Md-Ct		Metadata Traceability Md-Tr	
Strategic St	Strategic Taxonomy St-Tx	Strategic Structure St-Sr	Strategic Connectivity St-Cn	-	Strategic States St-St	-			Strategic Constraints St-Ct	Strategic Deployment, St-Rm Strategic Phasing St-Rm	Strategic Traceability St-Tr	
Operational Op	Operational Taxonomy Op-Tx	Operational Structure Op-Sr	Operational Connectivity Op-Cn	Operational Processes Op-Pr	Operational States Op-St	Operational Interaction Scenarios Op-Is			Operational Constraints Op-Ct	-	Operational Traceability Op-Tr	
Services Sv	Service Taxonomy Sv-Tx	Service Structure Sv-Sr	Service Connectivity Sv-Cn	Service Processes Sv-Pr	Service States Sv-St	Service Interaction Scenarios Sv-Is			Service Constraints Sv-Ct	Service Roadmap Sv-Rm	Service Traceability Sv-Tr	
Personnel Pr	Personnel Taxonomy Pr-Tx	Personnel Structure Pr-Sr	Personnel Connectivity Pr-Cn	Personnel Processes Pr-Pr	Personnel States Pr-St	Personnel Interaction Scenarios Pr-Is			Competence, Drivers, Performance Pr-Ct	Personnel Availability, Personnel Evolution, Personnel Forecast Pr-Rm	Personnel Traceability Pr-Tr	
Resources Rs	Resource Taxonomy Rs-Tx	Resource Structure Rs-Sr	Resource Connectivity Rs-Cn	Resource Processes Rs-Pr	Resource States Rs-St	Resource Interaction Scenarios Rs-Is		Measurements Pm-Me	Resource Constraints Rs-Ct	Resource evolution, Resource forecast Rs-Rm	Resource Traceability Rs-Tr	
Security Sc	Security Taxonomy Sc-Tx	Security Structure Sc-Sr	Security Connectivity Sc-Cn	Security Processes Sc-Pr	-	-			Security Constraints Sc-Ct	-	Security Traceability Sc-Tr	
Projects Pj	Project Taxonomy Pj-Tx	Project Structure Pj-Sr	Project Connectivity Pj-Cn	Project Process Pj-Pr	-	-			-	Project Roadmap Pj-Rm	Project Traceability Pj-Tr	
Standards Sd	Standard Taxonomy Sd-Tx	Standards Structure Sd-Sr	-	-	-	-			-	Standards Roadmap Sd-Rm	Standards Traceability Sd-Tr	
Actuals Resources Ar		Actual Resources Structure, Ar-Sr	Actual Resources Connectivity, Ar-Cn	Simulation			Dictionary Dc Summary & Overview Sm-Ov Requirements Req		Parametric Execution/ Evaluation ^b	-	-	
12												

2.1.4.4 Future vision of systems engineering

Recently, International Council on Systems Engineering (INCOSE) published a communication that outlined the future vision for systems engineering [86]. That document outlined imperatives for systems engineering, including concepts of composable design, meaning that systems and their components will be able to be reconfigured in a simulation environment.

Composable design methods in a virtual environment support rapid, agile and evolvable designs of families of products. By combining formal models from a library of component, reference architecture, and other context models, different system alternatives can be quickly compared and probabilistically evaluated [86, p. 34].

Consistent with realizing these goals, Friedenthal [21] identified a the growth of capabilities of MBSE to include a defined ontology, and formalism such that these elements lead to model repositories that can be accessed for future use. This list is shown in Table 2.5, with the INCOSE growth path projected from 2010 to 2025.

Table 2.5. INCOSE MBSE roadmap, from 2010 to 2025. Adapted from [21].

• Emerging MBSE standards	• Defined MBSE theory, ontology, and formalisms
• Matured MBSE methods and metrics	• Distributed and secure model repositories crossing multiple domains
• Integrated system/HW/SW models	
• Architecture model integrated with simulation, analysis, and visualization	

2.1.5 Ontology

The ontology structures the data types and components within the model. This section outlines relevant structures, languages, and descriptions of these ontologies.

2.1.5.1 Basic formal ontology

This section outlines a particular ontology developed by Arp et al. [87] that could be used as a basis to categorize large sets of data structures. This work was a follow on to that of Smith [88] who developed an approach to semantics that applies directly to system architectures. The approach concentrates on the Basic Formal Ontology (BFO), originally developed for the medical research and applied to the human genome project [89].

Arp et al. derived the BFO with two fundamental categories of entities: *continuant* and *occurent*, as outlined in Figures 2.5 and 2.6 and defined by Arp [87] et al. with direct citations as follows:

Continuants: entities that *continue* or *persist* through time, including (1) independent objects (for example, things such as you and me); (2) dependent continuants, including qualities (such as your temperature and my height), and functions (such as the function of this switch to turn on this light); together with (3) the spatial regions of these entities occupy at any given time [87, p. 87].

Occurents: entities that *occur* or *happen*, variously referred to as “events” or “processes” or “happenings,” which we take to comprise not only (1) the processes that unfold in successive phases but also (2) the boundaries or thresholds at the beginnings or ends of such processes, as well as (3) the temporal and spatiotemporal regions in which these processes occur [87, p. 87].

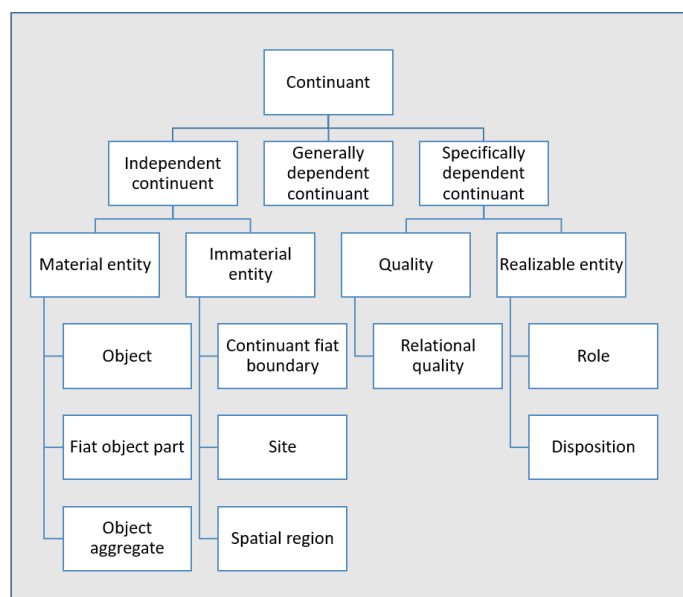


Figure 2.5. Hierarchy of Basic Formal Ontology (BFO) continuants. Adapted from [87, p.88].

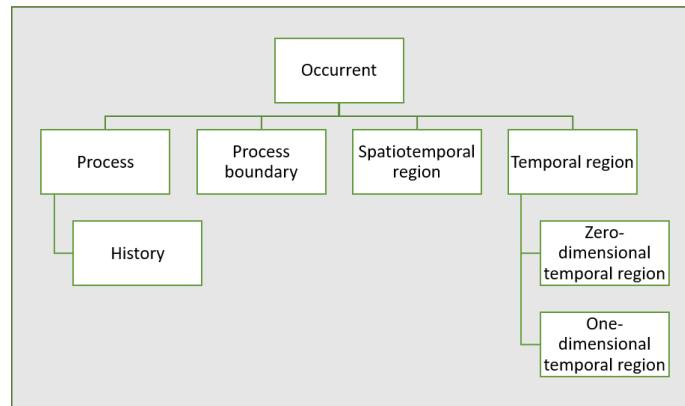


Figure 2.6. Hierarchy of Basic Formal Ontology (BFO) occurrents. Adapted from [87, p.128].

2.1.5.2 Ontology languages

2.1.5.2.1 Ontology layers: Fensel [90], [91] and Decker [92] developed the Ontology Inference Layer (OIL) concept. Davies [93] described the approach more formally.

2.1.5.2.2 Agent-based language: Using the groundwork of OIL, the Defense Advanced Research Project Agency (DARPA) Agent Markup Language (DAML) [94] provided a Department of Defense (DoD) representation of an ontology language.

2.1.5.2.3 Ontology language: The Web Ontology Language (OWL) [95] is a logical language, outlined by McGuinness et al. [96], with reference documentation by Bechhofer et al. [97], and the further documented by Antoniou [98]. OWL combines attributes of OIL and DAML in order to create a widely applicable ontology structure. The resulting data structure enables Structured Query Language (SQL) query of the data.

2.1.5.3 Ontology patterns

Gangemi and Presutti [99] developed the construction of Conceptual Ontology Design Pattern (CODEP) which is a form of an Ontology Pattern (OP) such that “natural to domain experts and laymen, i.e., the level at which small, expertise-aware components can be assembled as easy-to-apply, easy-to-customize building blocks” [100] by employing the OWL language and associated syntax. Gangemi and Presutti employed a structure derived from Alexander’s work, but adapted more general meta-data types of the following:

- *General issue* or title, often the class of the data object

- *Use case example* identifying a typical application
- *Notation* identifying the syntax definitions
- *Approaches* indicating the extent of the data
- *Considerations* indicating limitations and assumptions
- *OWL code listing*

Presutti [101] proposed a methodology to further the application of domain specific content Ontology Design Patterns (ODPs) while maintaining consistency with the general construction of the ontology. Conceptually, Presutti identified two simple steps: (1) of *matching* the current problems with already resolved solutions, and (2) of *selecting* a particular solution that addresses each particular problem.

Implementation of semantics within a consistent ontology became instrumental to conducting search algorithms [92], [93], [102], [103]. Applying the concept of semantics to pattern searches of behavior model results became a critical finding of this dissertation.

2.1.5.4 Patterns in system architectures

As an example of applying pattern definition to a particular domain, Alexander et al. [104], [105], developed a pattern language that describes towns, buildings, and construction such that the organization of the structural elements enabled insight to the future application of the design elements. Alexander posed each pattern, describing the problem, its context, and solution. He also related the concept of patterns having both a larger context into which the pattern fits and a smaller context such that more detailed patterns fit into it. Alexander's approach produced a hierarchy of elements, each with a consistent construct - following a pattern that led to a comprehensive description of 253 physical elements of design. This description formed the basis of a model that describes the set of physical elements. The concept of a pattern language lies at the heart of how to describe a model such that it relates to the ontology of a model and it forms the basis of the model definition.

Consider this extended description, that defines how Alexander documented each pattern:

For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explain how it helps to complete certain larger patterns. Then there are three diamonds to mark the beginning of the problem. After the diamonds there is a headline, in bold

type. This headline gives the essence of the problem in one or two sentences. After the headline comes the body of the problem. This is the longest section. It describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then, again in bold type, like the headline, is the solution – the heart of the pattern – which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction – so that you know exactly what you need to do, to build the pattern. Then after the solution, there is a diagram, which shows the solution in the form of a diagram, with labels to indicate its main components.

After the diagram, another three diagrams, to show that the main body of the pattern is finished. And finally, after the diamonds, there is a paragraph which ties the pattern to all those smaller patterns in the language, which are needed to complete this pattern, to embellish it, to fill it out. [104, pp. x–xi]

Gamma et al. derived a general pattern as applied to object-oriented software modeling, as indicated by Table 2.6.

Table 2.6. General pattern definition. Adapted from [104] and [106, p. 3].

Element	Description
The pattern name	Implementing a naming convention at a higher level of abstraction.
The problem	Describe a list of conditions where it makes sense to implement the pattern.
The solution	Describe the solution in a general sense, such that it is applicable to many situations.
The consequences	Describe the trade-offs and cost of applying the pattern.

Schindel [107] developed a methodology for Pattern-Based Systems Engineering (PBSE) that employs a generalized meta-model, or model of models, that is applicable across MBSE implementations. Schindel proposes that traditional systems engineering emphasizes an iterative process that leads to product realization; however, the systems engineering community must shift to an emphasis on the information that passes through the engineering process. Developing an understanding of the patterns intrinsic to the system supports this concept of an emphasis on the information. These

concepts also rely upon finding the smallest representation of a system [108]. As illustrated by Figure 2.7, he proposed that the progression from an abstract system description to a concrete product includes (1) a general system pattern, (2) product lines or system families, and (3) individual products or system configurations. The intent is to develop the general pattern implementing MBSE tools, particularly SysML.

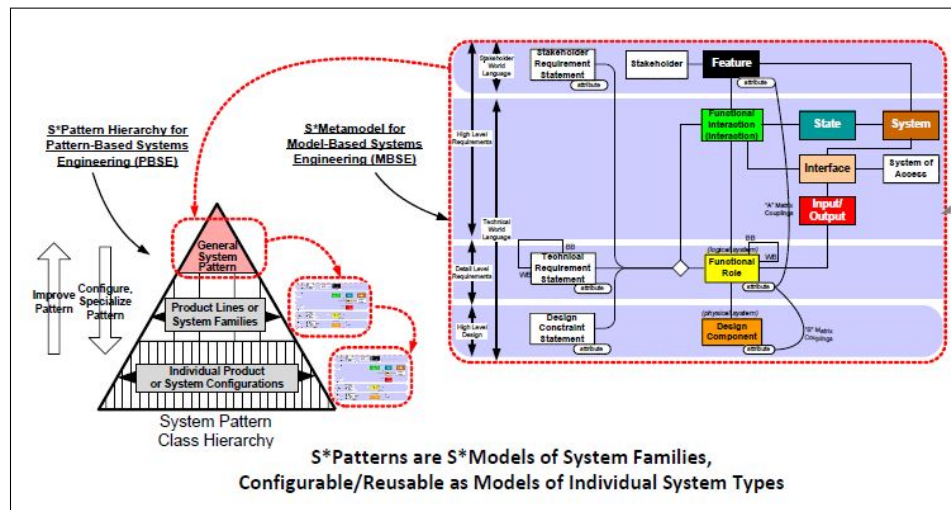


Figure 2.7. Pattern meta-model. Source: [107].

Schindel [107] also developed a cost model for implementing patterns within systems engineering, as illustrated in Figure 2.8. The top part of this figure illustrates the comparative return on investment for developing a pattern-based approach. MBSE has an inherently greater return since these structures can serve as a starting point for a families of products; however developing more general patterns enables a wider benefit across multiple families of products. As indicated in the lower part of the figure, the PBSE approach also reduces the cost per project since there exists a sound starting point for the new tasking. Conversely, using a previously employed MBSE project directly may have increased cost over starting that new project from a “clean sheet.” It is also interesting to realize that there is a cost to support both MBSE and the PBSE approach. These costs include training, tools, labor, and governance.

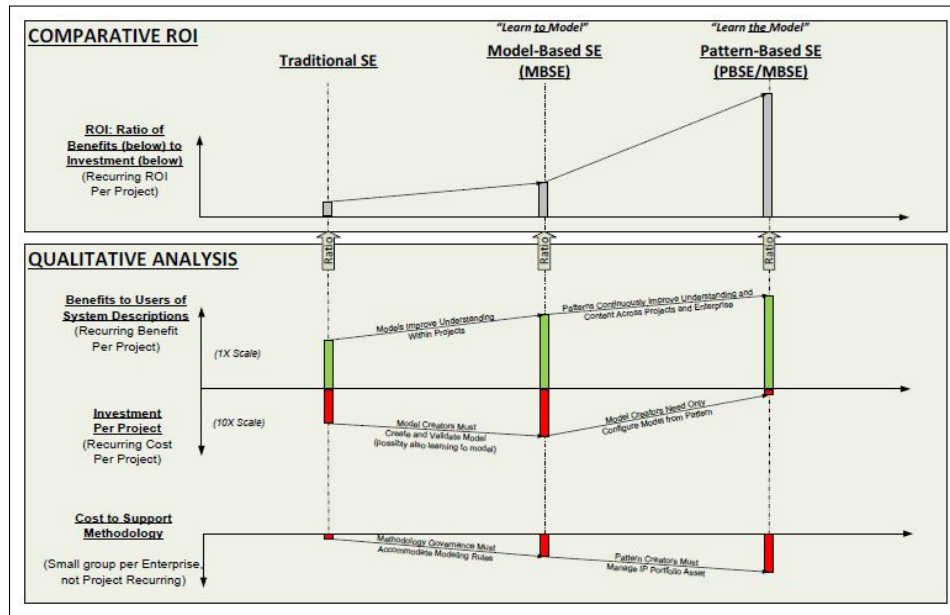


Figure 2.8. Cost of implementing systems engineering patterns. Source: Schindel [107].

2.1.6 Formal Methods

Zeigler et al. applied formal methods to systems modeling [109] and extended this construct to system of systems [110]. Formal methods enable an abstract description of a system, yet capture the definition in a way that is rigorous and mathematically composable [111]–[113].

The following description of formal methods was derived from the author’s previously published article [4].

Lightweight formal methods gain the advantage of applying a concise specification by implementing formal language in its description without incurring the cost of formal proofs [113], [114]. A formal specification of the model is fundamental to enabling automated checking so that the developer can quickly investigate and change the model. Formal methods hold to the axioms of predicate logic

This approach enables system definition at a higher level of abstraction, before investment in detailed design.

Monterey Phoenix (MP) is a lightweight formal method built with the intent to model behaviors comprised of events and their interactions [11], [115]. Monterey Phoenix (MP) was inspired by the Small Scope Hypothesis and the associated Alloy Analyzer [12], [13] that models objects and their relations but not their behaviors. MP does

model behaviors of a system architecture, and it enables the developer to interact with the model definition, add interactions, apply constraints, and adjust the definition as needed. Behavior in MP is modeled as a set of events, making up an event trace, where an event is an abstraction of any detectable activity, task, or component. Incorporating a concise language, MP employs event grammar rules. Within these grammar rules, the schema, or code, separates the definition of the behavior sequence from the definition of the interactions among the behaviors [66].

Two basic relations are defined for events: precedence and inclusion [66]. Precedence defines the ordering of events, enabling a description of structured order while also allowing concurrency. Inclusion defines the hierarchy of events, stemming from a root event that is logically distinct from other events and yet it may interact with other events. A root event is comprised of atomic events that are singular in description, or composite events that contain multiple atomic events and/or additional composite events. Interactions among behaviors allow sharing of an event and coordination of events.

2.1.6.1 Executable behavior architectures

Monterey Phoenix (MP) affords a means to build behavior models, using formal methods, that are adaptable to systems engineering, and extensible to levels of abstraction suitable for the engineering and design of systems of systems [10], [11]. These structures, coded with a formal language, are then executed to form a comprehensive series of traces, each representing a unique series of events.

MP defines individual component behavior as events and separately allocates the coordination of behaviors among components in the system and the external elements. These external entities may include the environment, user, or external systems. Executing MP produces a series of traces. Each of these traces is considered as a use case, or possible resulting behavior of the system. The developer defines the scope that limits MP execution.

Developing the architecture, by defining behaviors at a relatively high level abstraction, enables system definition of essential elements of the system while avoiding the minute details and constraints of its implementation.

Focused emphasis on analysis of the architecture affords the opportunity to consider intrinsic aspects of the system under consideration, without the need for designing the details of data structures, software language, executable code, or implementation. Likewise, deferral of hardware aspects

of particular design solution enables concise definition of the requirements and interfaces prior to detailed design of physical components such as transmitters, receivers, sensors, and network design. In this way, investigation of behaviors of a system, both positive and negative, occur before large expenses of the resources needed to implement the system. Formal analysis at a high level of abstraction also affords the developer a sound theoretical framework to derive requirements of the system.

2.1.7 Digital Engineering (DE)

Zimmerman outlined core aspects of the ongoing initiatives of digital transformation across the Department of Defense [116]. These principles led to the DE strategy, published in 2018 [117]. The strategic goals include the following:

1. Formalize the development, integration, and use of models to inform enterprise and program decision making...
2. Provide an enduring authoritative source of truth...
3. Incorporate technological innovation to improve the engineering practice...
4. Establish a supporting infrastructure and environments to perform activities, collaborate, and communicate across stakeholders...
5. Transform a culture and workforce that adopts and supports digital engineering across the life cycle [116].

The DoD, Systems Engineering Research Center (SERC), and Naval Air Systems Command (NAVAIR) pursued a consistent strategy through the ongoing initiatives of the Systems Engineering Transformation (SET) [118]. Blackburn et al. supported the SET by developing a practical means to enable coordination of MBSE models through Open Model Based Engineering Environment (OpenMBEE) [119], [120]. The intent of the Digital Transformation (DT) strategies across the services is to renew the commitment of the technical and acquisition communities to achieve real-time maintenance of the technical baseline of developing capabilities. The SET has sought to fundamentally change the workforce, tools, and infrastructure to enable efficient and continuous inquiry of systems under development.

These objectives are being instituted as the way of doing business for all future procurement. As outlined by the Army Acquisition Support Center, a pilot credential program has been initiated to develop formal education of DE through the Defense Acquisition University (DAU). Objectives include the following:

- Understanding the role of model-based systems engineering,
- Determining the need for digital artifacts-related standards,
- Defining a finite set of digital artifacts,
- Developing constructs for assembling digital artifacts, [121].

This dissertation seeks to employ the fundamental concepts of DE by conducting early analysis of system behaviors prior to extensive investment of development and acquisition resources. Future programs would benefit by ensuring that the systems under contract are fundamentally sound, with a concise description of the system behaviors. The formal language outlined in this dissertation explicitly defines the system behaviors, the system to system interactions, and the necessary constraints that must be employed by the operational capability.

2.1.8 Systems Engineering (SE) Resources

Resources for systems engineering within the Department of Defense (DoD), academia, and industry include the following:

- *The Defense Acquisition Guide (DAG)* as maintained by the Defense Acquisition University, Ft. Belvoir, Virginia:
<https://www.dau.mil/tools/dag>
- *The Systems Engineering Body of Knowledge (SEBoK)* site as governed by the Systems Engineering Research Center (SERC), the International Council on Systems Engineering (INCOSE), and the Institute of Electrical and Electronics Engineers Computer Society (IEEE-CS).
[http://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_\(SEBoK\)](http://sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK))
- *The Office of the Deputy Assistant Secretary of Defense (ODASD) for Systems Engineering* site includes policy & guidance, resources, and initiatives.
<http://www.acq.osd.mil/se/index.html>
- *The Defense Technical Information Center (DTIC)* also serves as a repository for documents, many of which are publicly released and available on-line.
<http://www.dtic.mil/dtic/>
- *The Consortium for Robotics and Unmanned Systems Education and Research (CRUSER)*

provides an on-line for ethical robotics research, managed by the Naval Postgraduate School.
<http://my.nps.edu/web/CRUSER/>

- *Monterey Phoenix (MP)* exercises behaviors and interactions of a system, enabling early analysis of emergent properties of the system. The on-line tool is an integral part of the proposed research.
<https://wiki.nps.edu/display/MP/Monterey+Phoenix+Home>
<https://firebird.nps.edu/>
- *Systems Engineering Research Center (SERC)* is a University-Affiliated Research Center (UARC), supporting the Department of Defense (DoD), ephasizing collaboration within the systems engineering, led by Stevens Institute of Technology and the University of Southern California (USC).
<https://sercuarc.org/>
- *Open Model Based Engineering Environment (OpenMBEE)* facilitates an on-line reference to engineering practitioners, including an Model Development Kit (mdk), Model Management System (mms), and View Editor (ve)
<http://www.openmbee.org/>

2.2 Assessment of Previous Work

The context of this dissertation encompasses the derivation of behavior patterns of designed systems of systems. The foundation of Model-Based Systems Engineering (MBSE), and particularly the construct of SysML and associated tools has served the systems engineering practitioners reasonably well. As with any developing field, however, interconnected and interdependent culture develops, and as technologies make more advances possible, practitioners of systems engineering need to adapt in a way that encourages solutions that have a wide range of application. The previous section outlines relevant literature, and some level of editorial comments were made for each of the topics outlined. This section outlines a more general assessment. Consider the following areas of interest, discussing engineering discipline, ontologies, patterns, and behaviors:

2.2.1 Engineering Context

Most engineering tasks are associated with directly supporting a product or program of record with a specific update or new capability. Because of this nature, it is difficult for engineering practitioners to develop architecture structures that are developed at a higher level of abstraction so that the model can be used by multiple products across domains.

2.2.2 Challenges of the MBSE Vision

The projected goal of incorporating formalism within a systems engineering modeling language may address an inherent disadvantage of DoDAF [86]. From its initiation, systems engineering practitioners began to use UML, a software developer's tool, as a means to precisely illustrate the DoDAF viewpoints. As SysML developed further, it provided utility to capture requirements, object parameters, and other aspects of the system important to engineers. These capabilities provided the systems engineering community a great benefit since the corresponding relational database supports configuration management, user-based query, and representation of the model in a way that is relevant to the stakeholders. However, the DoDAF viewpoint definitions were developed by consensus, based on the perceived needs, not in a way that ensures that the model is logically or mathematically complete. Therefore, the developer must be careful to ensure consistency throughout the model.

Employing a formalism to SysML models and architecture practice remains challenging; however formal definitions and mathematically composable models may best address the concerns of Ang et al. [75] and Mercer [76] in order to avoid the situation of being compliant with multiple viewpoints while remaining incomplete or poorly composed. Mercer posits that there “is insufficient formalism or scientific foundation behind most architecting practice within the DoD and part of this deficiency is attributable to the architecture frameworks used in DoD architecting.” [76, p. 558].

Up until this point, the available research has applied formal methods to specific problems within separate associated domains. Attempting to apply formal methods to a complete system of systems is quite daunting since it relies upon aligning all aspects of a domain to communicate in the same language, using the same semantics across all aspects of the model that is employed with a common ontology, employing a simple syntax that is easily understood, and done in a precise way such that logical reasoning holds together and mathematical calculations are possible. The systems engineering practice has not yet enabled a comprehensive approach to accomplish integration at this level.

2.2.3 Ontologies and Semantics

Model reuse relies upon implementing semantics that correspond to a defined ontology. The semantics relate the attributes of the model events and interfaces to the meaning of these model elements. The concepts employed by Web Ontology Language (OWL) [99]–[101] and Basic Formal Ontology (BFO) [87], [89] need to be employed consistently across all model domains that intend to share or reuse architectures.

2.2.4 Patterns and Reuse

The application of modeling language methods as applied to systems engineering has been derived from the techniques of computer science and has transitioned to solve systems engineering problems. Consider the transition from Unified Modeling Language (UML) to Systems Modeling Language (SysML) as a demonstration of that approach. At first, systems engineering practitioners observed the benefits of capturing the software within an object-modeling method, and applied those tools, at first directly, to designed systems development. Gamma indicated the importance of developing patterns with a level of abstraction so that they can apply to multiple problems [106]. Shindel developed a methodology that takes advantage of the concept of abstraction, as it could be applied to MBSE, and particularly to SysML.

2.2.5 Pattern Concepts Applied to SoS Behaviors

In order to formulate a means of pattern identification and reuse of the underlying architectures of designed system of systems behaviors, reliance upon previous work has become paramount. Prior research outlined extensive contributions that apply pattern-finding techniques to software, using light-weight formal methods to identify all possible sets of outcomes within scope [12]. Contemporary research is at the infancy of applying this approach to system behaviors [11], [122]. This dissertation seeks to derive a method that identifies underlying constructs of system of system behaviors, derive the associated patterns, and illustrate a means to reuse those patterns through applying semantics within a common ontology.

2.3 Definition of Terms

2.3.1 Systems

This section outlines basic distinctions among systems, Systems of Systems (SoSs), Complex Adaptive System (CAS), and their resulting emergent properties.

2.3.1.1 A system

The International Organization for Standardization (ISO) defines a *system* as “a combination of interacting elements organized to achieve one or more stated purposes” [123], where the elements may include “hardware, software, firmware, people, information, techniques, facilities, services, related natural artifacts and other support elements” [124]. *Environment* refers to factors that influence a user, system, or other entity and are within a common domain. *Interaction* is the activity of an entity with respect to another entity.

2.3.1.2 Systems of Systems (SoSs)

Jamshidi emphasizes the design goal of a designed system of systems with the following definition: “Systems of systems are large-scale integrated systems which are heterogeneous and independently operable on their own, but are networked together for a common goal” [125]. Organized by types of control as directed, acknowledged, collaborative, or virtual, as initially outlined by Maier [126], [127] and further elaborated upon by Dahmann, [128]. These concepts have been employed in an operational context [129]. A short summary of these definitions is as follows:

- *A System of systems* refers to a group of systems that interact with one another for a specific purpose.
- *A Constituent system* refers to a system that makes up part of the system of systems.
- *Collective Behavior* is the response of a group of systems, users, and the environment within which they interact.

2.3.1.2.1 Complex Adaptive Systems (CASs) J. Holland described Complex Adaptive Systems (CASs) as those systems that are “characterized by evolution, aggregate behavior, and anticipation” [130], and further described as “systems that involve many components that adapt or learn as they interact” [131].

2.3.1.2.2 Emergence is an observable result produced by interaction among two or more systems, users, or a combination thereof, within a common environment, and within an applicable domain; these behaviors cannot be deduced by investigation of an individual entity in isolation from the group [71], [132].

2.3.1.2.3 Emergent Behavior refers to the observable outcome of the interactions of the subject system or system of systems.

O. T. Holland derived a taxonomy of emergence [56] as a means to identify emergent behavior from other types of observable actions among systems. He describes emergence as “some phenomena

whereby quantitative interactions between entities in a system can lead to qualitative changes in that system which are different from, and irreducible to, the entities comprising it.”

2.3.2 Models

The scope of this research encompasses computational models of systems behaviors. Within this scope, critical elements include logic, abstraction, models, simulation, syntax, semantics, and ontology discussed in further detail as follows:

2.3.2.1 Abstraction, models, and simulation

Once again, let us start with the Merriam-Webster dictionary results for definitions of abstract, models, and simulation, again only the relevant definitions are listed as follows:

abstract [133]

1a: disassociated from any specific instance - an abstract entity

model [134]

3: structural design - a home on the model of an old farmhouse

4: a usually miniature representation of something - a plastic model of the human heart;

also: a pattern of something to be made

5: an example for imitation or emulation...

11: a description or analogy used to help visualize something (such as an atom) that cannot be directly observed

12: a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs;

also: a computer simulation (see simulation 3a) based on such a system climate models...

simulate [135]

2: to make a simulation of (something, such as a physical system)

simulation [136]

1: the act or process of simulating... 3a: the imitative representation of the functioning of one system or process by means of the functioning of another - a computer simulation of an industrial process b: examination of a problem often not subject to direct experimentation by means of a simulating device

The concept of an abstraction is critical to any discussion of a model. Simply put, a model represents only those functions necessary for a particular purpose; the model is a representation,

not the actual entity. As an example, a training system may need to instantiate a radio in order to provide instruction to aviators. However, the representation of the radio does not need to perform the thousands of functions of an operational radio; it only simulates a finite number of instances of the operational system to the user.

Many are familiar with the popular quote from Box that “all models are wrong, but some are useful” [62]. Buede developed a consistent thought as he defines a **model** as “any incomplete representation of reality, an abstraction” [64, p.75]. He further elaborates that a model may take on the following forms:

- *a physical representation* such as an aircraft wind tunnel model, used to identify aerodynamic properties to be projected to the final configuration,
- *a mathematical representation* such as a random number generator or a physics-based simulation of aerodynamic loads, and
- *a mental representation* that may or may not agree with a physical representation, a mathematical representation, or estimation of reality.

Buede’s key point is that the “*essence of a model is the question or set of questions that the model can reliably answer*” [64]. Though a model is an incomplete abstraction of reality, yet the model must enable a ***complete set of answers to the questions of interest***. Thus in order to represent behaviors, a model needs to identify all aspects of these behaviors include both positive and negative interactions.

Consistent with those definitions, consider the following: “A model is a representation of something. It captures not all attributes of the represented thing, but rather only those seeming relevant. The model is created for a certain purpose and stakeholders” [137].

A **simulation** is any particular instance of a behavior model. Using the previous examples, a simulation of the wind tunnel aerodynamic model is one run of the wind tunnel, consisting of the initial set-up description and final results. The simulation of a physics-based computer model is one execution, based on a set of inputs, and the computed results. As we shall investigate behavior models in the next section, these too have particular results based on execution of the model.

Key points:

1. A model is an abstraction of the physical or instantiated system, developed in order to represent essential elements of the design.
2. The model must represent a complete set of behaviors, and analytically derive a means to enable intended behaviors, while restricting unintended behaviors.
3. A simulation is any instance of behavior extracted from the model.

2.3.2.2 Syntax, semantics and ontology

Syntax, semantics and ontology dictate the description of the model content, the meaning of the content, and the relationships among the content.

2.3.2.2.1 Syntax Beginning with the English language definition:

syntax [138]

1a : the way in which linguistic elements (such as words) are put together to form constituents (such as phrases or clauses) b : the part of grammar dealing with this 2 : a connected or orderly system : harmonious arrangement of parts or elements

In the context of modeling and simulation, syntax is the set of rules that dictate both text and graphical representation of symbols that correctly communicates the model content.

2.3.2.2.2 Semantics identify the meaning of an item in the model. Given the context of an event within a behavior model, the meaning of the event may be applied as an attribute of that event.

semantic [139]

1 : of or relating to meaning in language

2.3.2.2.3 Ontology describes the relationship of the kinds of things that make up the model. Considered as an abstraction of semantics, an ontology provides a structure within which to construct consistent hierarchies of entities in relationship to each other. An ontology defines similar attributes among the different kinds of things described within the particular domain of interest. An example of an ontology is the biological taxonomy of living things, as defined within a hierarchy of domain, kingdom, phylum, class, order, family, genus, and species represents an ontology that is widely accepted, defining ways in which we classify living organisms.

ontology [140]

1 : a branch of metaphysics concerned with the nature and relations of being

Ontology deals with abstract entities.

2 : a particular theory about the nature of being or the kinds of things that have existence

2.3.2.3 Composability

While “composability” does not have a dictionary definition, its use has been employed across the technical domains. Applicable to modeling and simulation for the purpose of test, evaluation, and training systems, Petty and Weisel compiled the following four definitions from the literature, and then offered the fifth definition as listed below.

The ability to rapidly configure, initialize, and test an exercise by logically assembling a simulation from a pool of reusable components [141].

The ability to create, configure, initialize, test, and validate an exercise by logically assembling a unique simulation execution from a pool of reusable system components in order to meet a specific set of objectives [142].

The ability to build new things from existing pieces [143].

The ability to compose models/modules across a variety of application domains, levels of resolution and time scales [144].

The capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements... Composability is more than just the ability to put simulations together from parts; it is the ability to combine and recombine, to configure and reconfigure, sets of parts from those available into different simulation systems to meet different needs. [145].

Davis and Anderson then developed the following more concise sixth definition listed below.

The capability to select and assemble components in various combinations to satisfy specific user requirements meaningfully. A defining characteristic of composability is the ability to combine and recombine components into different systems for different purposes [146].

Note that the first two definitions include the concept of a “logical” assembly of components, that may be inferred, but missing from the latter definitions. All definitions also include a concept of re-use of existing parts or components, and so orderly re-use of such components implies some existence of a repository that is built in a way that is consistent within some domain of all possible combinations of models.

Beginning with these definitions, and adding specific aspects of composability addressed in Chapter 2, consider that in order to be useful, systems engineering analytical methods, tools, and models need to be composable in the following ways: mathematically, logically, syntactically, and semantically.

Consider the following aspects of composability:

- *Mathematical composability* ensures the existence of a mathematical solution.
- *Logical composability* ensures sound reasoning of the model.
- *Syntactic composability* ensures consistent notation and usage of language.
- *Semantic composability* ensures consistent meaning of the entities within the model.
- *Ontological composability* ensures consistent structure of the meaning of the elements of the model.

With editorial adjustments, consider the following definition:

Key point:

Composability is the ability to employ components of a systems model that meet specific user requirements while maintaining mathematical, logical, syntactic, semantic, and ontological consistency.

2.3.3 Behavior

In order to present a cohesive representation of a system architecture in this chapter, a significant discussion on the concepts of abstraction, behavior, activity, events, modeling, and simulation become necessary. These definitions may be somewhat unique to the proposed methodology, but since our area of interest of a system of systems demonstrating positive emergence.

2.3.3.1 Behavior, activities, actions, and events

Monterey Phoenix (MP), developed by Auguston [10], [11], models behavior as a set of events and is proposed as a means to develop a system of systems architecture. A behavior relies upon an

operator performing a particular action on some object or operand. This action is considered an event. The following paragraphs list definitions for behavior, activity, action, and event from the Merriam-Webster dictionary, with alternate uses removed for clarity, as follows:

behavior [147]

- 1b: anything that an organism does involving action and response to stimulation
- c: the response of an individual, group, or species to its environment ...
- 3 : the way in which something functions or operates ...

From this definition, a behavior involves an action in response to some stimulation.

activity [148]

- 1 : the quality or state of being active : behavior or actions of a particular kind...
- 2 : vigorous or energetic action ...
- 3 : natural or normal function...

An activity relates to some set of actions or functions.

action [149]

- 1b: the accomplishment of a thing usually over a period of time, in stages, or with the possibility of repetition
- c actions plural : behavior, conduct...
- 2 : an act of will...
- 3 : the bringing about of an alteration by force or through a natural agency - the action of water on rocks...
- 7a: an operating mechanism
- b: the manner in which a mechanism or instrument operates - a drill's twisting action...

Here, we see that an action brings about temporal relationships, including some period of time, implying a beginning and an end.

event [150]

- 1c: a postulated outcome, condition, or eventuality...
- 2a: something that happens : occurrence...
- 5 : a subset of the possible outcomes of an experiment

Starting with these general definitions, concise uses of these terms need to be employed in order to develop the concepts of behaviors within some technical or engineering construct. To that end, let us first define a **behavior** as a set of events or actions. Behaviors are typically represented by multiple systems interacting with the environment and among the systems. There is also typically some sort of observable outcome (e.g., formation flight).

The **temporal** relationships among events is of critical importance to a behavior. A particular event or any be detectable or observable entity has a beginning and an end. This allows us to then discuss the concept of *precedence*. Precedence enables ordering of multiple events, since a particular event may occur before an other, establishing a precedence relationship. Alternatively, multiple events may occur concurrently or simultaneously, without a precedence relationship. This lack of precedence is also a temporal property, but in this case, the events are independent of each other. Both the existence of a precedence relationship or the non-existence of a precedence relationship are temporal attributes of an event.

An event also has an additional relationship, *inclusion*. This relates the hierarchy of events, such that all event must have a root or source. One may think of a radio that would either send or receive a message. The event under consideration is the send or receive action, the operator is the radio, and the operand is the message. And so one can say that the send and receive events stem from, or are included in, the radio. Complex events may be described within multiple subordinate events that stem from a composite event. For example, there may be many functions necessary for the radio to send or receive, and so the operator and operand remain the same, but the send and receive events may be made up of many sub-events. Within this chapter, actions and events are treated synonymously, though when used in reference to the MP code, the term “event” will be used exclusively. More simply, an activity is a higher level abstraction of an event. Following a defined process is an example of an activity.

Defining an event as particular instance of an activity with an observable beginning and ending within the time domain differs from the definition employed by a discrete-event simulation. A discrete event model represents the activities, or actions, that occur from event to event. And so in this case, an event is considered as some identifiable end point in the model.

Key points:

1. A **behavior** is a set of events or actions, typically leading to some observable end point.
2. An **event** has a beginning, and end, and an order of precedence. Both the existence of a precedence relationship and the lack of a precedence relationship (or concurrency) are relevant temporal relationships among events.
3. An **event** also has an additional relation, inclusion, indicating the hierarchy among events.
4. An **action** is considered the same as an event.
5. An **activity** is a higher level abstraction of an event. A process is an activity. For example an activity of “car driving” includes the events of “turn the wheel, depress the accelerator, and depress the brake.”

2.3.3.2 A behavior model

We have established that a behavior is a set of events or actions leading to some observable end-point, and a model is an abstraction of the physical or instantiated system, developed in order to represent essential elements of the design.

A behavior model in the analytical or computational realm consists of an algorithm that employs a formal language to derive a set of events, representing essential interactions and relationships held within the system. This approach avoids the specification of parameters of the design for as long as possible, enabling concentrated effort to evaluate the underlying and intrinsic interactions within the system and among interactions with other systems, the user, or the environment. The temporal and hierarchical properties formulate the essence of the behavior model.

A formal language, based in set theory, is fundamental to the description of a behavior model. As opposed to a natural language, with possibilities for misinterpretation, a formal language is logically complete, consistent, and verifiable. Formal methods are based on propositional and predicate logic, such that the former deals with the validity of a premise and conclusion, while the latter adds the expressions of “all, no, and some” [151], [152]. Monterey Phoenix (MP) employs such a language, explicitly for the purpose of defining and modeling behaviors. As a light-weight formal method [113], [114], models developed within MP rely upon intrinsic investigation by the developer for verification in lieu of formal proofs. This approach encourages an interactive methodology for development, outlined later in this dissertation.

Key points:

1. A **behavior model** employs a formal language to derive a set of events, representing essential interactions and relationships within the design.
2. This analysis is performed at a level of abstraction that isolates the logical behavior of the design.

In execution, MP generates all possible outcomes of the model. Within MP, these instances are called traces (or use cases) and are based on the scope of execution of the model. Simple models contain relatively few interactions and few alternative possibilities; and so a measure of complexity may be associated with the number of interactions and number of traces of the model.

MP employs the Small Scope Hypothesis developed by Jackson, positing that most software failures can be found with relatively few iterations of the code, as follows:

Most bugs have small counterexamples.

That is, if an assertion is invalid, it probably has a small counterexample... it has an encouraging implication: if you examine all small cases, you're likely to find a counterexample [12, p.143].

Modeling objects, as opposed to behaviors, Jackson also encoded this concept through the Alloy Analyzer [13]. The Alloy Analyzer (available at <http://alloytools.org/>) provides a means to execute object modeling structures.

Within the domain of behavior modeling, the scope of execution of the model increases the number of iterations associated with a particular behavior. For example, consider a send-receive sequence. A single iteration would execute as a single-scope as follows:

send-receive

Likewise, an additional iteration would execute as a scope of two, as follows:

send-receive
send-receive

An application of the small-scope hypothesis would be that any errors in the the model that produces this sequence would likely be found with only a few iterations of the sequence.

Monterey Phoenix (MP) provides an analytical platform for the developer to systematically define the events, add temporal and hierarchical interactions, apply constraints, evaluate the results, and adjust the model as needed. This dissertation develops a methodology and demonstrates a means to exhaustively evaluate all outcomes by evaluating patterns behavior in the model, thereby systematically evaluating all outcomes of the model, at a given scope of execution.

Incorporating a concise language, MP employs event grammar rules to define the model. Within these grammar rules, the schema, or code, separates the definition of the behavior sequence of a system from the definition of the interactions among systems. In this way, MP uniquely separates the definition of the behaviors within a system from the definition of the interactions of the events among multiple systems, the user, an environment [66]. This construct allows the model developer flexibility and control in describing the intended behavior.

2.3.4 System Processes and Patterns

In order to identify patterns, processes executed by the associated systems form the building blocks for the system pattern. The definition of a process is consistent for both natural or biological systems as well as those associated with designed systems.

process [153]

2a(1) : a natural phenomenon marked by gradual changes that lead toward a particular result

the process of growth

(2) : a continuing natural or biological activity or function such life processes as breathing

b : a series of actions or operations conducing to an end

especially : a continuous operation or treatment especially in manufacture

From these definitions, and relevant to systems under development, let us consider a process as a set of events that move an object form one state to another state.

Furthering this concept, the term **isomorphic**: literally means of the ‘same form.’ As applied to an Isomorphic Systems Process (ISP), referring to a process that is consistent across many domains [32], [34].

isomorphic [154]

1a : being of identical or similar form, shape, or structure

isomorphic crystals

b : having sporophytic and gametophytic generations alike in size and shape

2 : related by an isomorphism

isomorphic mathematical rings

Friendshuh and Troncale [32] define a a system process as a “series of steps typical of surviving systems that adequately fulfills a needed system function when considered at the abstract system level” [32].

pattern [155]

1 : a form or model proposed for imitation...

2 : something designed or used as a model for making things

...

4 : a natural or chance configuration

frost patterns

the pattern of events

...

7 : a reliable sample of traits, acts, tendencies, or other observable characteristics of a person, group, or institution

a behavior pattern

spending patterns

the prevailing pattern of speech

...

10 : a discernible coherent system based on the intended interrelationship of component parts foreign policy patterns

...

Alexander authored a foundational work on a pattern language, written within the context of architecture related to towns, buildings, and construction [104], [105]. Alexander writes that “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice” [104, p. x].

Within the context of object modeling and analysis, Fowler provides a general definition of a pattern as “an idea that has been useful in one practical context and will probably be useful in others” [156, p. xv]. Gamma et al. provides a more specific definition as patterns apply to design as

“descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.” [106, p. 3].

We see from these definitions that the intent of a pattern in design is to describe it in such a way as to use it over and over, and so some minimum content is necessary to describe the pattern. With these aspects in mind, consider the following as a working definition of a pattern within the context of describing system behavior.

Key point:

1. A **pattern** is a description that can be employed over and over again.
2. A **behavior pattern** is defined as a set of two or more events within a hierarchical and temporal structure that captures the path taken to an observable end point.
3. A **behavior template** is software query that identifies the pattern, capturing the topology and semantics of the pattern in such a way that allows reuse.

2.3.5 Summary of Terms

Listed alphabetically, the following provides a summary of terms that were previously described in this section:

- **Activity:** An abstraction or generalization of an event, something that is done. [148].
- **Atomic event:** An event that is not decomposed by other events. Applied in the context of MP [66].
- **Attribute:** A parameter associated with an event.
- **Behavior:** A set of one or more events, including interactions, that lead to some observable end point.
- **Behavior pattern:** A set of two or more events within a hierarchical and temporal structure that captures a path taken to reach an observable end point. The events within the pattern may be concurrent, chronological, or consecutive.
- **Chronological events:** Two or more time-dependent events such that one must occur before the other.
- **Collective Behavior** is the response of a group of systems, users, and the environment within which they interact.
- **Composite event:** An event comprised of other events. Applied in the context of MP [66].

- **Complex Adaptive System (CAS):** Systems of systems, typically consisting of many elements that adapt and learn as they interact [131].
- **Composability** is the ability to employ components of a systems model that meet specific user requirements while maintaining mathematical, logical, syntactic, semantic, and ontological consistency.
- **Concurrent events:** Two or more time-independent events.
- **Consecutive or Sequential Events:** Two or more time-dependent events such that one immediately precedes the other.
- **Constituent system** refers to a system that makes up part of the system of systems.
- **Emergence** is an observable behavior produced by interaction among systems, users, and a common environment, within an applicable domain; these behaviors cannot be deduced by investigation of an individual entity in isolation from the group [71], [132].
- **Entity:** An object, activity, concept, or other item that exists. An event is an entity, but not all entities are events.
- **Environment** refers to factors that influence a user, system, or other entity and are within a common domain.
- **Event:** An activity with an observable beginning and ending, existing within the time domain. An event may be instantaneous or occur within a finite time frame [66].
- **Formal methods:** A model specification that employs a logically-precise language, derived in a way that is consistent with axioms of propositional and predicate logic [151], [152].
- **Interaction** is the activity of an entity with respect to another entity.
- **Isomorphic:** literally means of the ‘same form’ [34].
- **Isomorphic Systems Process (ISP):** Refers to a consistent or universal process of systems from many domains [32], [34].
- **Lightweight formal methods:** Formal method employed without proofs [13], [113], [114].
- **Process:** A set of events that move an object from one state to another state.
- **Root event:** An event without a predecessor, typically characterizing the function of a system, user, or environment. Applied in the context of MP [66].
- **Ontology:** An organized structure that describes the nature of entities or objects [140].
- **Pattern:** a description that can be employed over and over again.
- **Semantics:** The meaning of an entity [139].
- **System** is “a combination of interacting elements organized to achieve one or more stated purposes” [123], where the elements may include “hardware, software, firmware, people, information, techniques, facilities, services, related natural artifacts and other support elements” [124].

- **System of systems** is a group of systems that interact with one another for a specific purpose [157].
- **Template:** A computational query that encompasses a description of the behavior pattern, including the topology and semantics.
- **Topology:** Properties of the geometric form of a pattern, as represented by nodes and edges of a network. For a behavior model, the nodes represent events and the edges represent directed relationships [158].
- **Trace:** A particular sequence of events produced by executing the model. Applied in the context of MP [66].

CHAPTER 3:

Research Method

The dissertation research employs a design-based method using an analytical framework that exposes inherent behaviors of a system of systems, as captured in the architecture description. The approach relies upon formal definition of the architecture, as indicated by Monterey Phoenix (MP).

3.1 Research Design

The proposed design-based research is facilitated by developing a algorithm that automatically investigates MP output and identifies patterns inherent in the results. The research method, outlined in Table 3.1, includes the design of this algorithm and application to several Monterey Phoenix (MP) models. This serves as a means of verification of the utility of the algorithm and applicability of the approach.

3.1.1 Pattern Detection

Of course, the SE process [159] was used to develop a algorithm to automate behavior pattern detection in MP: requirements development, functional analysis, synthesis of the model, and verification. Table 3.2 outlines the design attributes of the pattern detection method.

The subject behavior analysis includes evaluation of the following:

- *Necessary sequences* that must occur within the system for proper function, including behavior of the system and coordination with the environment and adjacent systems.
- *Failed sequences* that may occur within the system, lacking sufficient fail-safe degradation of critical systems. Failed sequences may be a form of negative emergence (e.g., obstacle collision, network or memory contention, or suspended execution).
- *Functions* of the system develop from the logical architecture of the system, and must be reflected in the system requirements.

Table 3.1. Research method.

Category	Description
Research type	Design of an automated means to assist the developer of a system of systems to understand the range of behavior responses of the system under development.
Contribution	The contribution to the body of knowledge in systems engineering provides a methodology and automated means to identify inherent behavior patterns in the associated system of systems architectures.
Purpose	System of systems architectures produce hundreds or thousands of use cases that cannot be adequately assessed without automated assistance to the developer.
Products	1. A general methodology to develop and evaluate system behaviors, 2. An automated algorithm that finds patterns inherent in the system architecture, 3. Graphical output of results, 4. Behavior model schemata, 5. Derivation of a belief network, consistent with the constraints to derive the probability of the set of possible behaviors of the model, 6. Derivation of a Design Structure Matrix (DSM), enabling an automated means to group behaviors.
Scope	The research products are intended for the developer of systems of systems, yet general enough for application to other domains.
Motivation	Investigation of system of system behaviors, both positive and negative, containing large sets of possible outcomes.

3.2 Data Sources, Collection, and Analysis

Data sources were generated digitally through modeling, simulation, and analysis tools. The effort implemented behavior modeling and simulation, including developing Monterey Phoenix (MP) schemata, Matlab code, and other software sources for illustration. The models concentrate on the domain of designed system of systems that may apply to Department of Defense context. Applicability of “designed” systems include not only material solutions, but also non-material systems. Sources of these data were derived from previously published work, as cited, and also generated as new concepts.

Table 3.2. Methodology design attributes.

Item	Description
Objective	Develop an automated algorithm to assist in the detection among of hundreds or thousands of event traces generated by MP.
Functions	1. Automatically find patterns intrinsic to the event traces, 2. Ensure that all patterns are reviewed, 3. Identify patterns of interest for assertion checking within MP, 4. Assess the scope of MP execution to determine whether any new interactions were identified at a higher scope. 5. Derive the probability of each outcome of the model to assist with evaluating the likelihood of occurrence of each pattern of interest.
Verification	1. Develop several MP schemata across several domains to ensure that the algorithm functions, and that it provides accurate and consistent results. 2. Derive a methodology that identifies intrinsic behavior patterns occurring in the architecture. 3. Test the algorithm to find known flaws in simple MP schemata, comparing the results of the automated algorithm with those of a manual analysis.

3.3 Research Concept

Figure 3.1 illustrates a generalized depiction of MP results, with two traces. Each trace contains both content that is unique to the individual trace and content that is common to both traces. Systematic review of the MP results is important to ensure that wanted behaviors exist in the model and also to ensure that unwanted but intrinsic behaviors in the architectures do not exist.

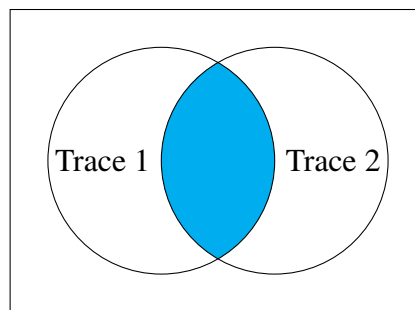


Figure 3.1. Identifying common patterns among traces.

Common content needs to be analyzed only once, and so identifying common content becomes relevant. Repeated review of similar output forces the developer to attempt an ad hoc approach for analysis, thereby increasing possibility of making mistakes and significantly delaying completion. An ad hoc approach is trivial for fewer than ten to twenty traces, but this approach becomes

inhibitive and potentially non-tractable with larger sets of output. Even simple models can produce thousands of traces, each representing a possible outcome of the system.

Finding inherent patterns in the system architecture is a means to identify common content across multiple traces. The following pattern search strategy was employed as a means of conducting the research.

3.3.1 Pattern Search Strategy

The pattern search strategy is as follows:

1. *Develop a basic behavior model* using MP,
2. *Define all relationships* that include hierarchical and temporal interactions of the events, capturing both the actions within each system and the interactions among systems,
3. *Derive constraints* that limit undesirable behaviors,
4. *Identify patterns* by querying the model for semantically aligned attributes and temporal relationships,
5. *Evaluate the results* by deriving the inherent Design Structure Matrix (DSM), the probability of occurrence of each trace, and the impact of constraints on the model.
6. *Enable re-use* by compiling a repository of semantically relevant search patterns that follow a unified ontology.

The following section provides more details of how to implement this strategy, including further explanations, discussion, and examples.

3.3.2 Elements of Behavior Patterns

As briefly discussed in Section 2.3, the pattern exists in the behavior model, and is intended as a representation of the system under development. Conversely, the template is a computational search query that captures critical elements of the model through the topology and semantics associated with the pattern. The topology is the node and edge structure of the pattern, that is represented by boxes and lines. The meaning of the boxes and lines are captured by the associated semantics. These semantics include the interpretation of architecture: (1) the boxes represent the hierarchical elements and events of the model, (2) the solid lines represent a precedence relationship between the events, (3) the dotted lines represent an inclusion relationship between the events, and (4) attributes are associated with the events, the simplest of which is a Boolean representation (i.e., either true or false), illustrated as blue text.

Note: Throughout this chapter, all figures that were derived from MP employ the key shown in Figure 3.2, unless otherwise noted on the figure. An extended description of these elements is as follows:

- The *boxes or nodes* of the model represent a specific activity, considered as an event in MP. Representation of objects within the behavior model may be considered as the activity conducted by the object. Since it becomes a bit lengthy to write out the verb form of “conduct the action of an object” at each event node, the name of the object is often used as the event description. This practice is particularly useful in order to represent the hierarchical relationship of the subordinate events. The style used in this document identified the activities conducted by an object as a *noun* written as capitalized text (e.g., “Control Tower” or “Aircraft”). Conversely, the specific events were written as a *verb* and in lower-case text (e.g., “send message” or “receive message”).
 - A *Root event* is typically a representation of the activity conducted by the environment, a user, a system, or a component in the model that has no preceding or hierarchical element. A root event is displayed as a green box.
 - A *Composite event* contains subordinate elements, that may consist of events or attributes, displayed as an orange box.
 - An *Atomic event* has no subordinate elements, displayed as a blue box.
 - A *Note* is depicted by a faded-yellow box.
 - For the *illustration* of an event, a white box is used when referring to any event, whether root, composite, or atomic.
- The *lines or edges* represent relationships within the model. Within MP all of the relationships are directed, starting with a source and ending with a target.
 - An *inclusion* relationship is represented by a dashed-line and arrow. This is consistent with the SysML representation of a “has a” relationship. For example, as illustrated in Figure 3.3a, System A *has an* event a1.
 - A *precedence* relationship is represented by a solid-line and arrow. This may be considered as a “need-line” that identifies necessary source event(s) that must occur prior to the target event. If a target event has multiple sources, all source events must occur prior to the target event.
 - A *user-defined* relationship is represented by a blue solid-line and arrow.
- *Attributes* apply meaning or parametric data to the events. Boolean attributes were used to

derive behavior patterns, for example the event “send message” is a form of a transmit action and has the attribute of “Tx: true.” For this document, the attributes are listed as blue text.

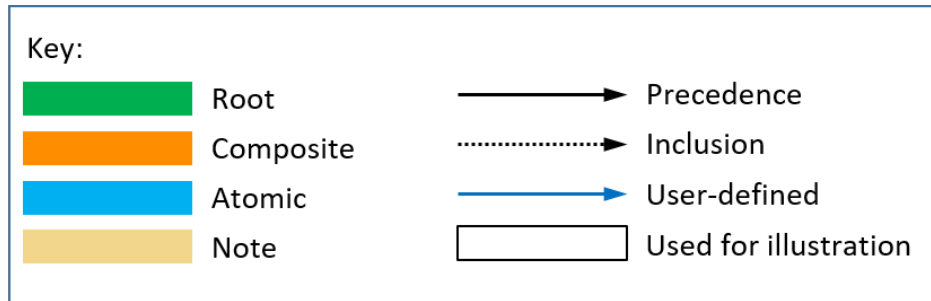
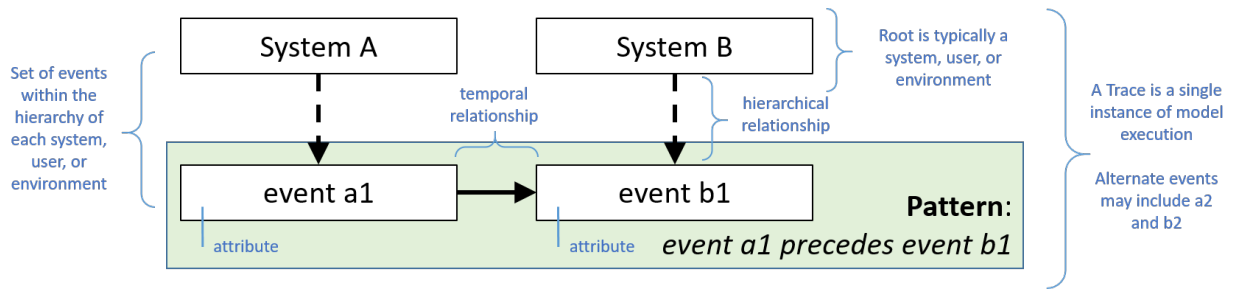


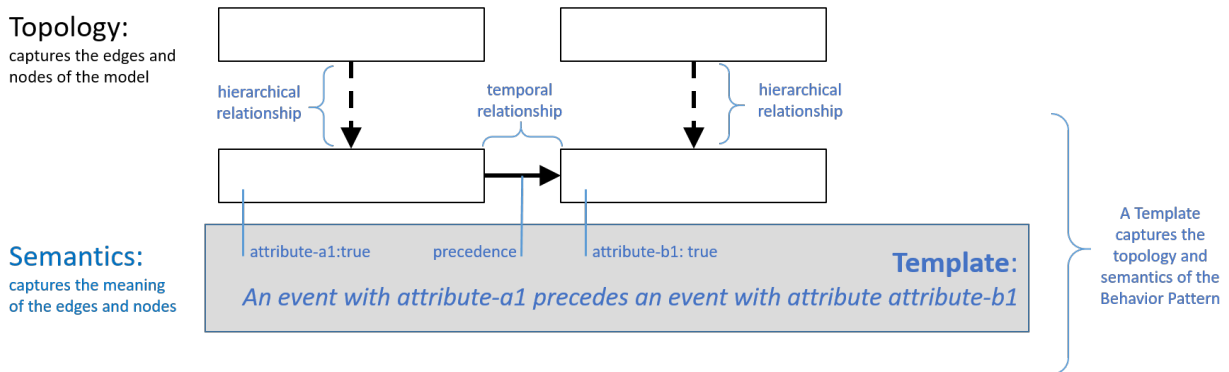
Figure 3.2. MP diagram key.

3.3.2.1 Derivation of a template reflecting a behavior pattern

A pattern is defined as a set of two or more events within an inclusion and precedence structure that captures the path taken to an observable end point. A template is software query that identifies the pattern, capturing the topology and semantics of the pattern. Figure 3.3a illustrates a basic pattern of two events such that event a1 “precedes” event b1 and the template illustrated in Figure 3.3b captures both the attributes of these events and the temporal relationship.



(a) Minimum pattern: two events and a relationship (hierarchical or temporal) between them.

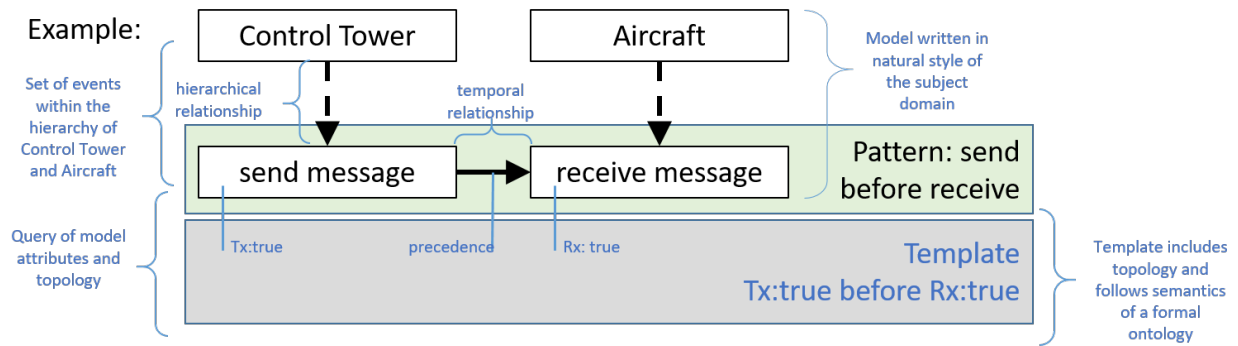


(b) Minimum template: attributes of each event and topology represented by the two events within a precedence relationship.

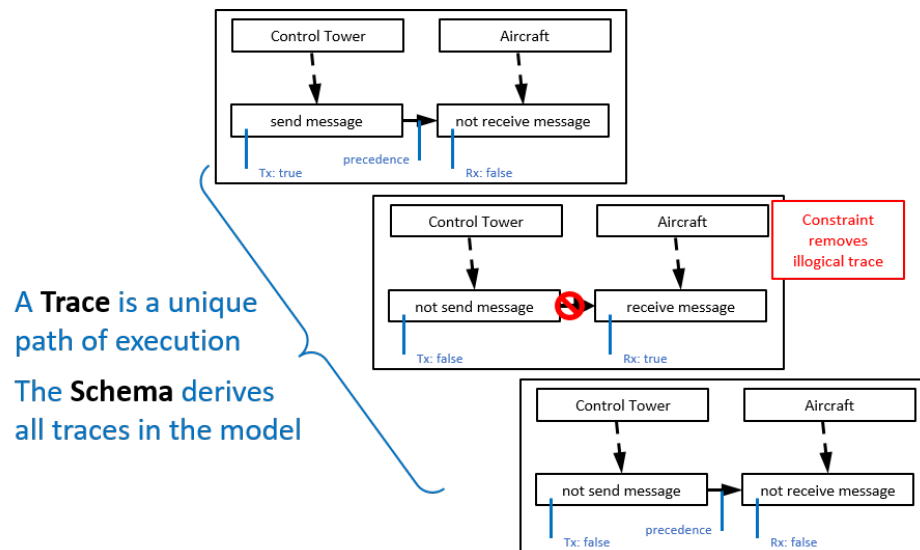
Figure 3.3. Behavior pattern and template definition.

3.3.2.2 A behavior pattern example

As illustrated in Figure 3.4 such that a control tower sends a message that is received by an aircraft. Alternatives include not sending a message and not receiving the message, creating four potential traces based on the combinations of these alternatives. A constraint must be imposed so that when a message is not sent, none can be received, leaving three valid traces within the schema. Figure 3.5 provides an example of a pattern that is met by the template.

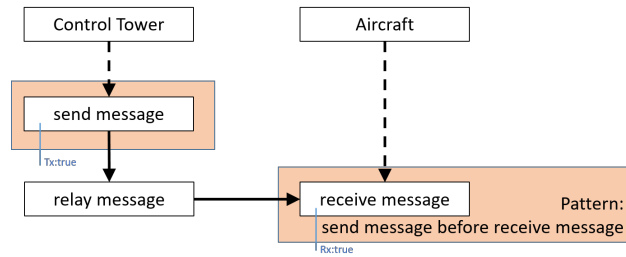


(a) Example pattern and associated template.

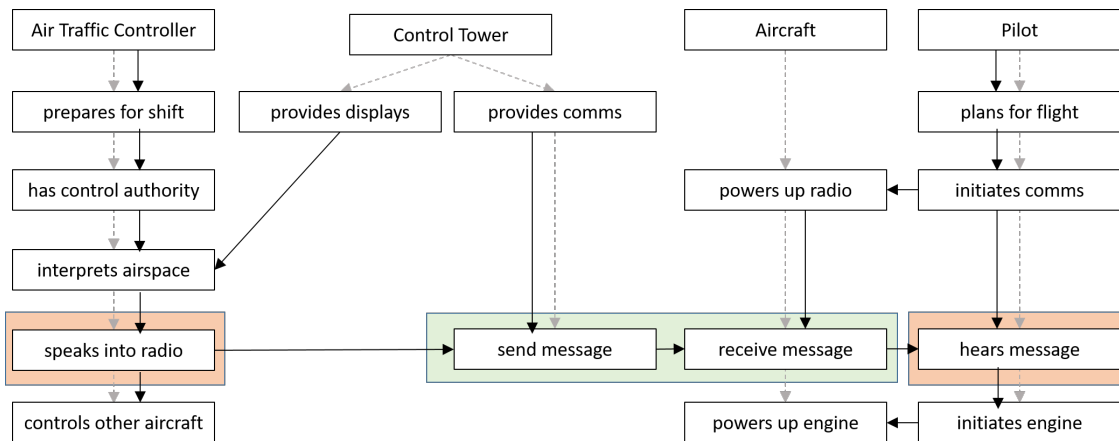


(b) Pattern search of alternate paths of execution. Each set of alternative events creates a unique trace. The constraint prunes the attribute combination $Tx: false$ precedes $Rx: true$ for the application instance of not send message precedes receive message. This constraint applies for any corresponding sending and receiving pair, following this logical constraint.

Figure 3.4. Behavior pattern example.



(a) Event “send message” is before but does not immediately precede event “receive message”



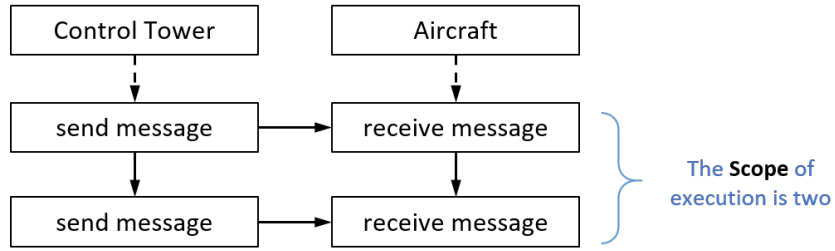
(b) Event “send message” precedes “receive message” and “speaks into radio” is before “hears message”

Figure 3.5. Examples of the pattern met, such that one event occurs *before* the other.

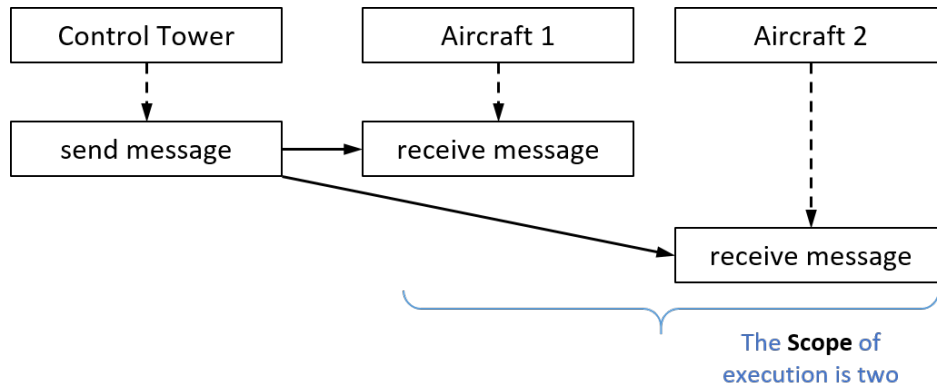
3.3.3 Scope of Execution

The scope of execution of a behavior model refers to the number of instances of an event or sequence of events within the model. As illustrated in Figure 3.6, this may take the form of two examples.

(1) multiple iterations of the same event, or (2) multiple objects with an instance of the event.



(a) Multiple iterations of the same set of events in the same hierarchy. Notice that the first receive message instance is only preceded by the first send message instance.



(b) Multiple instances of the event “receive message” in an expanded hierarchy.

Figure 3.6. Scope of Execution Example.

3.3.4 System Behaviors Compiled as a Directed Graph

An adjacency matrix, A , may be employed that compiles the interactions among events of each trace. The adjacency matrix is composed by populating all of the events as rows and columns of the matrix, with each row having an index of i and each column having an index of j . The directed relationships are indicated by the source event in row i to the target event in column j . Since both inclusion and precedence relationships have a source and a target, the adjacency matrix may represent only the inclusion relationships, only the the precedence relationships, or both the inclusion and precedence relationships. Each trace has an associated adjacency matrix. By summing the adjacency matrices identified by each trace, a cumulative matrix, $sumA$ can then be readily be computed such that all interactions are captured, across all traces associated with MP execution.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1j} \\ a_{21} & a_{22} & \cdots & a_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i1} & a_{i2} & \cdots & a_{ij} \end{pmatrix} \quad (3.1)$$

such that i and j are the integers from 1 to the number of uniquely named events,

a_{ij} is the number of precedence and/or inclusion relationships from event at index i as the source to the event at index j as the target, and

Figure 3.7 illustrates the concept of building a cumulative adjacency matrix, including a representation of multiple traces, patterns between these traces, and the corresponding adjacency matrices associated with each trace.

$$sum A = \sum_{n=1}^N A_n, \text{ where } N \text{ is the number of traces.} \quad (3.2)$$

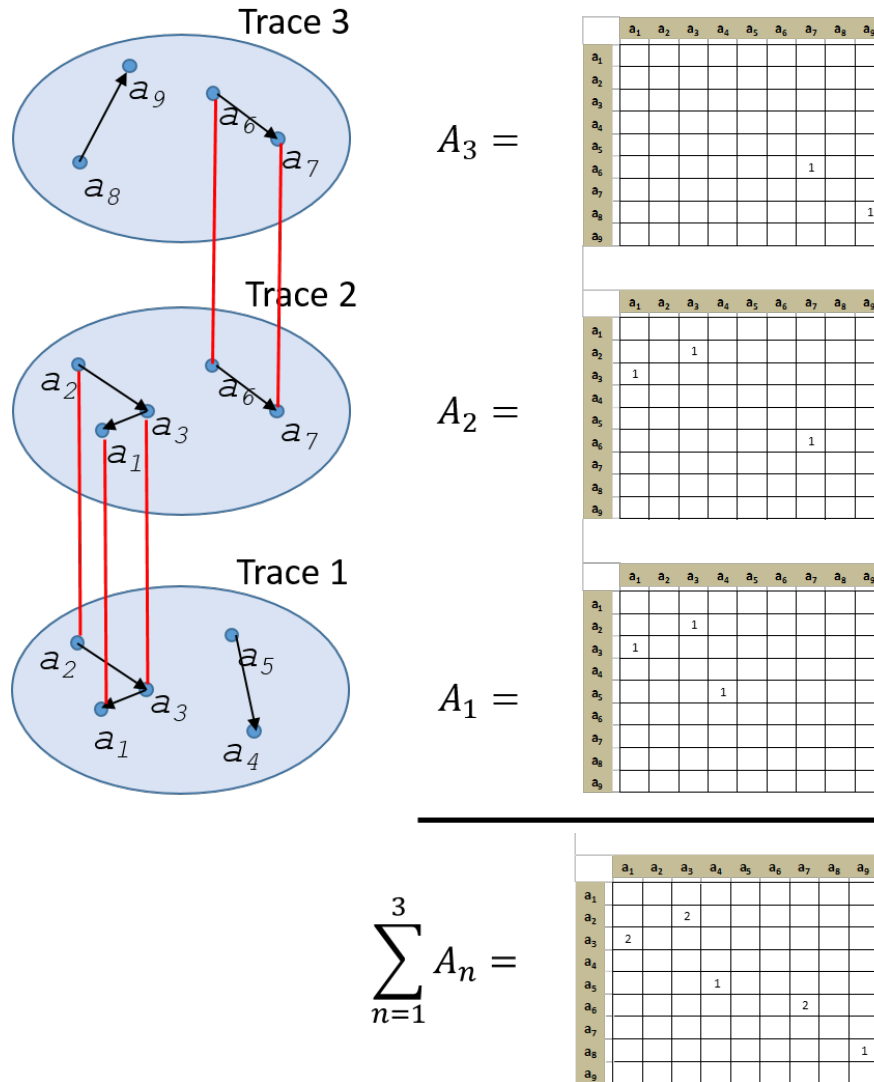


Figure 3.7. Conceptual example of patterns.

Figure 3.8 provides a simple example of the send and receive sequence of events, with a tabulation of all root and atomic events to form a cumulative adjacency matrix. This adjacency matrix composes the Design Structure Matrix (DSM) that captures all interactions among all traces of the behavior model. One use of the DSM is to compare model output while varying the scope of execution. If new interactions occur at increased scope, the developer can investigate these particular interactions. This comparison necessitates a post-execution environment to compare results from multiple instances of execution. The goal is for the SoS developer to use the adjacency matrix as a DSM and apply the associated algorithms used for systems development as outlined by Browning [160] and Eppinger and Browning [161]. This dissertation outlines a means to use MP

results, populate the DSM, and then employ a clustering algorithm, as derived by Thebeau [162], to automatically sort and order the rows and columns of the matrix. This clustering algorithm then derives potential opportunities modularize the SoS in ways that may be not immediately obvious to the developer.



Figure 3.8. Simple example of patterns, tabulated in the form of individual adjacency matrices of each trace, followed by a cumulative matrix as the associated sum. Each is displayed as a Design Structure Matrix (DSM) in this figure, with the name of each event listed in the rows and columns. The input-row (IR) has the source in the row and the target in the column.

3.4 Dissertation Contribution

Within a defined scope, MP exhaustively determines all possible traces in accordance with the system definition. The resulting traces include both unique and common patterns in the results. When developing reasonably intricate architectures, the results produced within this dissertation have produced hundreds or thousands of traces; however, examining these traces becomes impractical without a means to systematically interpret the results.

As discussed throughout, MP-Firebird was used as the main analytic tool, and as such the research

gained from particular capabilities inherent in the modeling language. The following list of attributes are provided in order to ensure that a distinction is made between MP and the dissertation contribution.

Attributes of Monterey Phoenix (MP):

- MP-Firebird employs a formal language such that the results are composable mathematically and logically.
- As a light-weight method, no proofs are provided, and so the user interacts with the tool through assertion checking to ensure the viability of the results.
- MP-Firebird employs propositional and predicate logic through a set of command statements and the associated syntax.
- MP-Firebird represents behaviors in the most fundamental form through the relationships of precedence and inclusion.
- MP-Firebird is executable and it develops instances of behavior that encompass all possible outcomes of the model, within the scope of execution.
- MP-Firebird provides a user interface and graphical output.

The main contribution of this dissertation is a methodology that enables the developer to describe the behaviors of a particular system, define the interaction of the system with other systems, users, and the environment, add constraints to the model, derive patterns from the set of possible outcomes, analyze and interpret the results, and re-use the model for other problems of interest. Attributes of this contribution include the following:

Key points on the dissertation contribution:

- All steps of the methodology are applicable to large data sets. The models demonstrated the methodology that evaluated up to tens of millions of potential outcomes, with thousands of valid outcomes that were compliant with applied constraints on the behavior.
- Prior to this dissertation, and the associated publications about the dissertation, there had not been a documented means to exhaustively identify intrinsic behavior patterns in system architectures in a way that could support a repository and model reuse based on event attributes and topology.
- This methodology defined a means to employ a set of consistent semantics within a defined ontology, so that the resulting query of the model is conducted through search templates that can be archived in a repository and retrieved for future use.
- The dissertation proposes a taxonomy for the model constraints: *logical, simplification, design, and definition*. These constraints and the associated interaction of events formally described requirements necessary for the system to function as intended. The constraints also refine the number of potential outcomes as set that is desirable and considered valid by the user.
- The dissertation delivered a post-processing script, *trimBranch.m*, written in MATLAB® that aligns a Bayesian belief network to the set of constraints applied to the model. This script operates on a fixed topology, with any number of events, and any number of alternatives for each event.
- The contribution includes the method, schemata, scripts, and applications as listed within the body of the dissertation, thus being available for follow-on research.

3.4.1 Templates Applied to Behavior Models

The next chapter provides a detailed derivation of each of the models, serving as a means to validate the methodology. However, before proceeding to that section, a discussion of the utility of the dissertation contribution is outlined in the following sections.

3.4.1.1 Repository archive of templates

The concept of reuse relies upon the existence of an archive from which the user can apply templates to a new project, or a revitalization of an existing project. And so a repository of templates that follow a consistent ontology would benefit multiple users.

The resulting methodology has been shown to identify the following:

- A description of baseline templates, such that at an increased scope the behaviors are based on a set of combinations these behaviors.
- A set of template segments that capture short sequences of events such as Observe, Orient, Decide, Act (OODA). These segments of events relate directly to the Isomorphic Systems Processes (ISPs) outlined in Section 2.1.2.
- All positive events that are considered as acceptable outcomes.
- All negative events that formulate a set of anti-patterns or “pathologies” associated with the model as conceptualized by Troncale [163].

3.4.1.2 An example template

In order to illustrate the examples listed above, first consider an abstract model of a decision structure. The contents of this model will be discussed in detail in the next chapter, but the model is illustrative of most of the key findings of the dissertation research.

Figure 3.9 illustrates a decision model abstraction of a template, indicating that each event is associated with its more generalized semantic designation, and those attributes, along with the topology form the template. Templates operate on the semantics and topology, not on the model written in the natural domain language of the developer. The developer simply aligns model events with semantics obtained from a repository of semantics.

In this example, illustrated in Figure 3.9, the template, shown on the right, becomes part of the repository so that other users can gain access to the archive at a future date. After applying the set of templates, the user may then decide whether to keep the same constraints of the original model, develop new ones, or discard some and allow more viable outcomes of the model. As illustrated in Figure 3.10, for the surgical model, only the names of the events changed. Conversely, in the aviation model, the names of the events changed and additional events were added, as illustrated in

blue. The template sorted through the models and found the same set of templates for each trace of execution, and then reported the results back to the user in the same domain language that would be expected.

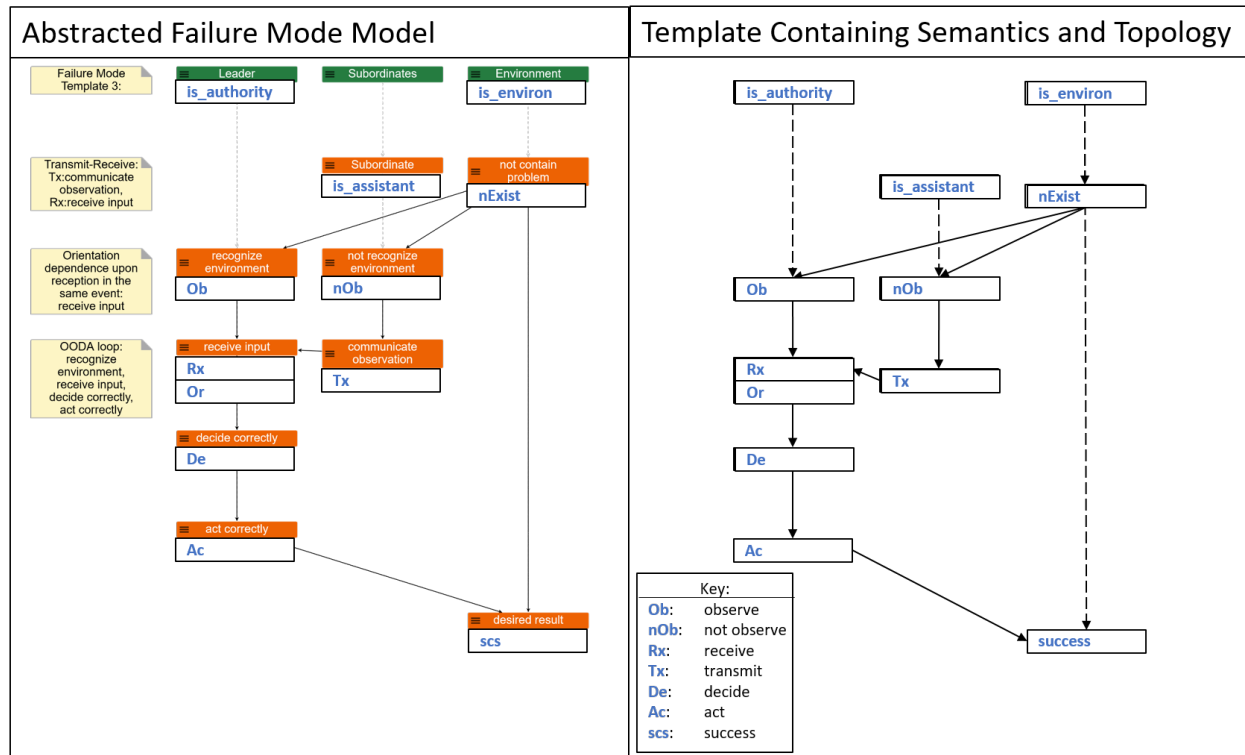


Figure 3.9. Decision Model template abstraction, indicating that each event is associated with its more generalized semantic designation, and those attributes, along with the topology form the template.

Figure 3.10 illustrates an example of how a user may employ the same template to two different models, and customize the model for readability and context to the user-domain nomenclature while maintaining the identical template by assigning attributes to each of the new model events.

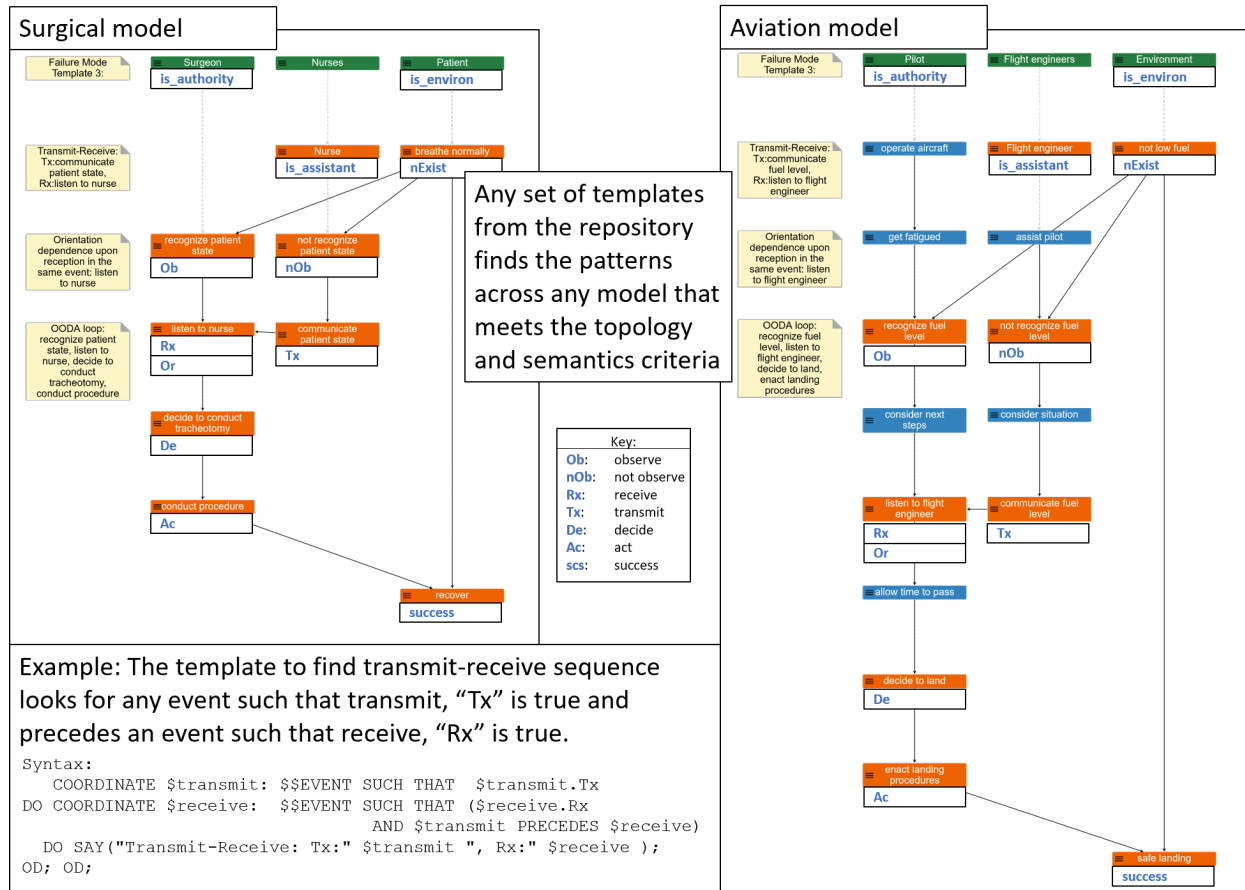


Figure 3.10. Decision Model domain-specific applications, employing identical search patterns because the attributes aligned to both models followed the same semantics. Notice that the pattern still identified each significant event, illustrated in orange, in order to complete the template. Other events, shown in blue, have no bearing on this template.

3.4.1.3 The Decision Model executed at an increased scope

Now, let us consider the effects of increasing the scope of execution such that the model now has two subordinates, as shown in Figure 3.11. This effect created two concurrent paths in the same trace of events. Both of these templates (Failure Mode Template #9 and #10) were found when the model was executed with only one subordinate, at scope equal to one. Those templates were kept in the repository, and without any modification, they correctly identified the two valid concurrent event segments. This illustrates the effect of increasing the scope of execution to two for this model, enabling two subordinates. Since both templates were initially defined at scope one, and defined the behaviors exhibited at scope two, the results provide an indication that the Small Scope Hypothesis holds in this case. In other words, both of these sequences of events were observed when the scope execution was set equal to one.

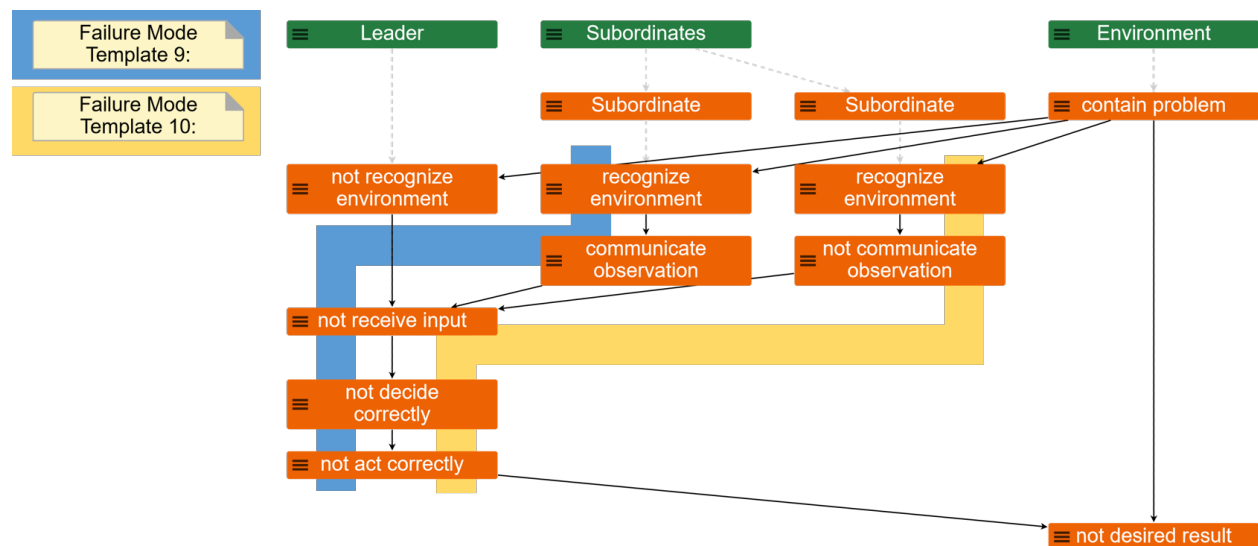


Figure 3.11. Decision Model with two subordinates, enabling an additional contribution to the decision process. Therefore two viable paths are illustrated in yellow and blue, tracing back to the initial templates that were derived by executing the model with only one subordinate.

3.4.1.4 The Decision Model with identified positive and negative events

Finally, this section includes an illustration of a pattern template that identifies the set of all positive and negative events associated with each trace. This template relies upon the attribute selection of each decision being either favorable or unfavorable. Later models also illustrate that these assignments can be conditional, based on previous events that have occurred. Figure 3.12 illustrates the utility of this effective and simple template.

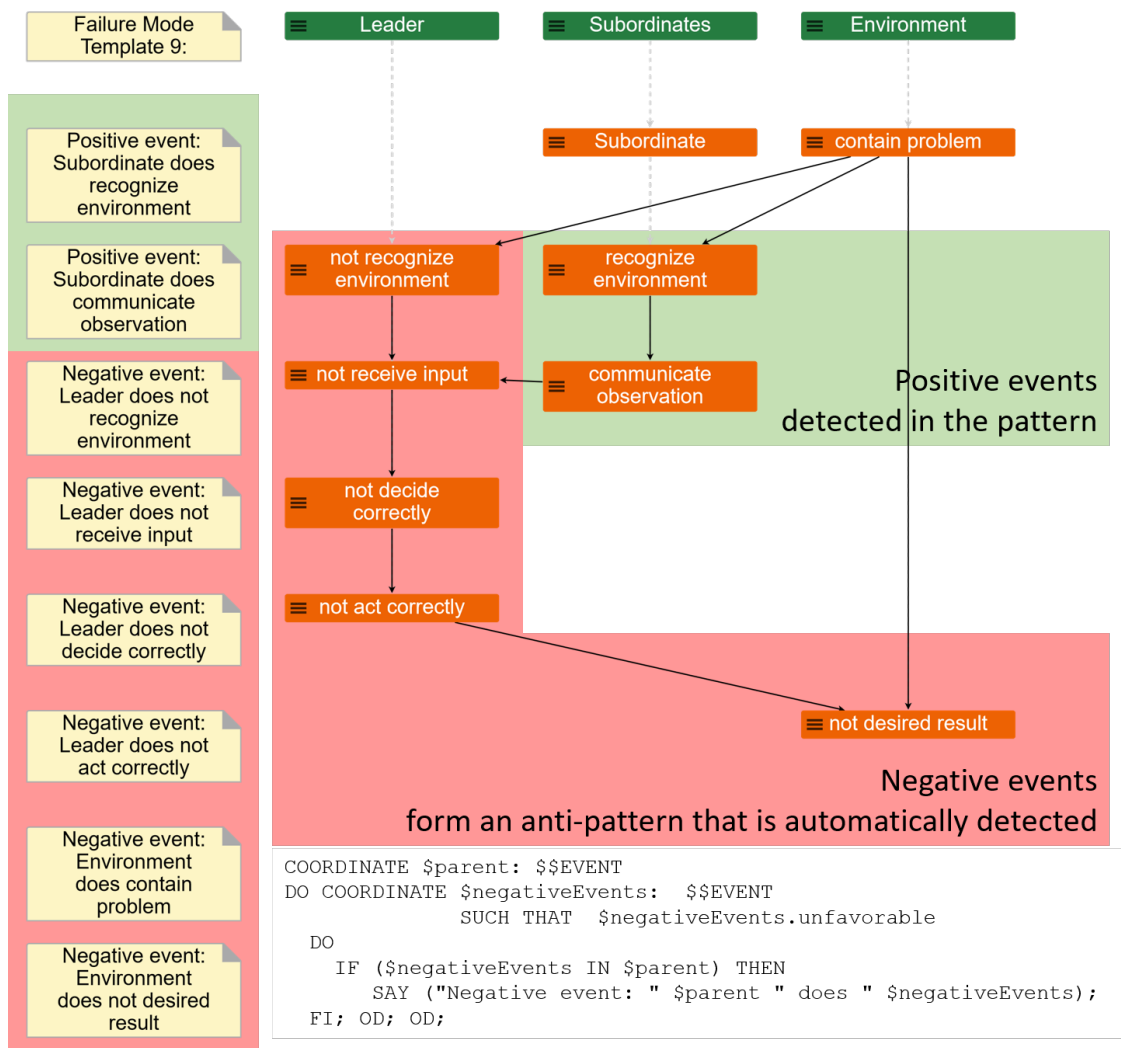


Figure 3.12. Decision Model identifying both positive and negative events.

CHAPTER 4:

Research Findings

This chapter describes the findings of the dissertation research. The first section outlines the methodology, followed by a detailed derivation of each of the models.

The models that were developed include the following:

1. *Example models*: were used to illustrate the methods, scripts, and sample results in a way that is readily described. These examples include applying the template search query to the cyclic ISP, simple probability examples, and derivation of the DSM.
2. *Decision Model*: Outlines a decision mode problem experienced in a hierarchical workforce, dealing with the perception of a routine procedure that leads to a critical problem. Inspired by a leadership book by Syed [17]. Details provided section 4.3.1, and the set of graphical output and code are listed in Appendix A.1.
3. *Object-to-Effect Model*: Reflects an operational task that relies upon multiple sensors to produce compiled information that is sufficient for some critical action. Derived from Dahmann et al. [18] and Antul et al. [19]. Details provided in section 4.3.2 and the set of graphical output and code are listed in Appendix A.2.

4.1 Methodology for Pattern Derivation

Monterey Phoenix (MP) [15], [164], [165] provides explicit definition of the system's behavior architecture and may thereby expose potential flaws in logic can show up by building and analyzing the architecture. The MP construct separates the descriptions of behavior within the system and the interaction among the systems, environment, and users, giving the developer flexibility in the design of the architecture.

The Small Scope Hypothesis, derived by Jackson [12], postulates that most errors in computer code can be found within a relatively small number of iterations or execution cycles. The small scope hypothesis and the associated Alloy Analyzer [13] provided inspiration to apply this practice from the software domain to the systems engineering domain with Monterey Phoenix and the MP-

Firebird tool [165]. This dissertation explores the applicability of this hypothesis as demonstrated through systems engineering problems. The results identified in Section 4.3 show that patterns experienced from models executed at higher scope of execution were combinations of patterns derived at a lower scope of execution.

The Methodology for pattern derivation does not simply evaluate a small portion of the set of possible outcomes and trust that the rest are valid. On the contrary, the MP analyzer always produces all of the possible outcomes, and the developer works with this set of outcomes in order to constrain unwanted sequences and ensure that wanted sequences are maintained. Pattern identification helps the developer to understand the content of the model, especially when advanced models produce large data sets of possible outcomes.

Key advantages of using MP for the methodology:

1. MP exhaustively identifies all possible outcomes of the behavior architecture, within the scope of execution. These results expose patterns of behavior leading to the the need for analytic tools to interactively identify these patterns.
2. MP capitalizes on the Small Scope Hypothesis [12] indicating that most inconsistencies in the model are found within relatively few iterations.
3. MP builds a model at a high level of abstraction, prior to detailed design and synthesis of the system, enabling the identification of problems early in the design process.
4. Evaluating the model with MP early in the development enables the use of a mature structure for follow-on simulations employing event-based models, agent-based models, time-based physical models including system dynamics, or hybrids of these methods.

This section provides an overview of the methodology to identify and control behavior patterns of a System of Systems (SoS), illustrated in Figure 4.1. Each step in the methodology has an associated interactive execution, a forward progression, and an option for re-work based on the results of that execution.

The methodology applies the following steps:

1. ***Develop the behavior narrative:*** This step relies heavily upon input of a subject matter expert in concert with the model developer. From the narrative, the developer derives the

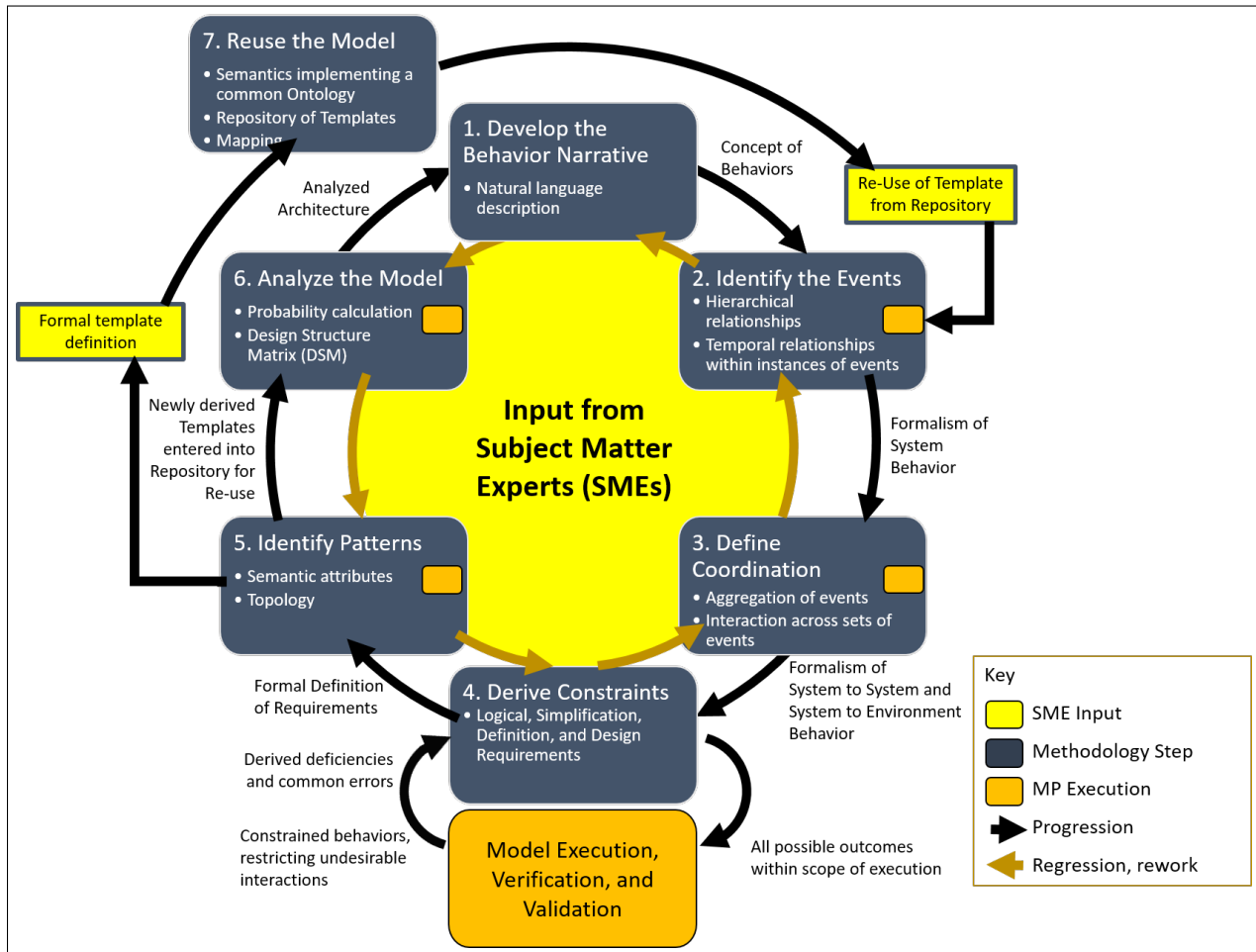


Figure 4.1. A methodology to develop the System of Systems (SoS) behavior architecture, conceptually derived from the software development spiral model [166]. Progression is illustrated by the black arrows from step 1 through 6. Regression, or rework, is illustrated by the tan arrows, and is based on the results of model execution. Adapted from [6].

main actors, the associated events identified by each actor, the sequence of events, and the interactions among the events. Of particular interest is for the subject matter expert to identify alternatives to the expected path of execution.

2. **Identify the events associated with each system:** From the narrative, the developer now formalizes each of the relevant events. This step translates the narrative to the building blocks of the behavior model, establishing the hierarchical (inclusion) and temporal (precedence) relationships each set of events. For an SoS, this step defines the events within each system.

3. **Define the coordination among the systems:** This step outlines the temporal (precedence),

and user-defined relationships of the events within the model. At the completion of this step, all possible patterns of execution are defined by the model. For an SoS, this step defines the external interaction from system to system.

4. ***Derive the constraints of the model:*** Given the potential alternatives at each event, execution of the model produces hundreds or even thousands of possible execution paths. The developer uses constraints within the model in order to enable the desirable execution paths and to limit undesirable execution paths. These constraints then become a set of requirements that are needed to produce expected results of the system under development. Types of constraints are listed as follows:

- *Logical* constraints limit the model from executing fallacious paths, such as receiving a message when none has been sent.
- *Design* constraints limit the model from executing paths in order to control the desired pattern of the system. For example, a system that incorporates multiple sensors may include a voting scheme based on the desired performance of the system.
- *Simplification* constraints limit the output by restraining alternate paths that are not of interest. For example, a user may always decide to initiate a sequence, and the model may eliminate the possibility of not initiating a particular sequence.
- *Definition* constraints limit the output by outlining a pattern such as success-failure criteria. As an example, a correct decision within the model may be considered as a success criteria, and so the constraint eliminates an incorrect decision.

Verification and validation of the model is closely related to the application of the constraints, and so the developer would be expected to conduct many iterations between these steps in order to ensure that the final model produces expected and manageable outcomes.

- *Verification:* In a general sense, verification ensures that the model was constructed in the correct way; therefore, much of the concentration of effort should be to ensure that the logic inherent in the model makes sense. Initially, most models are prone to have logical inconsistencies, and by inspection, the developer needs to ensure that the inconsistencies are removed. Examples of logical inconsistencies abound, and they are dependent upon the context and meaning of the model: a message cannot be received if not sent, a book cannot be read if not written, and a car can not be driven without a driver. The meaning of the event and the context within which it is used make up the *semantics* of the model. In this way, the semantics of the model are bound to the logic of the model.

- *Validation:* Also described here in a general sense, validation ensures that the model is useful; as such, the model, as an abstraction of the associated end item, accomplishes its intended purpose. And so at this step, interaction between the developer and the Subject Matter Expert (SME) provides an opportunity to ensure that the design constraints are effective, and that the simplifications and definitions are appropriate. Certain deficiencies of the model architecture may come to the forefront at this step, and so sections of the model may need to be re-constructed in order to address deficiencies.

Key point on the interactions and constraints:

The model interactions (step 3) and the associated constraints (step 4) establish a formal definition of system requirements, derived interactively with the model.

5. ***Identify the patterns within the behavior model:*** Ideally, the developer has now constrained the model from thousands of possible outcomes to tens of outcomes that form patterns of execution paths. These execution paths may contain both desirable and undesirable outcomes since it may not be possible to constrain all of the paths that may lead to a system failure or other undesirable result. The execution paths, referred to as traces, are able to be queried as templates. Both the topology and the semantics formulate the necessary aspects needed to establish a template and thereby systematically query the model. The templates identify the inherent patterns of behavior. The goal of the developer is to identify a set of templates, when used in combination, capture the all outcomes of the model executed at higher scope.

Key points on patterns:

- (a) Align model events to Boolean attributes based on the semantics available in accordance with the ontology associated with the domain.
- (b) Interactively develop new templates that employ the semantics and topology associated with the model.
- (c) Execute the model in order to identify the inherent patterns. The templates link the developer's interpretation of the model to an exhaustive software query of the all possible outcomes, exposing the underlying construct of the architecture. Using a query ensures that every pattern is found, even among thousands of potential results.
- (d) Interact with that model to ensure that all sets of outcomes accomplish the desired functions of the model. Since the templates were derived interactively from the behavior description, not super-imposed on the model of the system, this tends to

be a creative step in the methodology. These patterns may encompass all decision points of the model, or only a shorter segment of the decision model.

- (e) After the core patterns are defined, the user may elect to relax the simplification constraints, while keeping the logical, design, and definition constraints. This step enables the developer to sift through the additional traces now enabled again by the model.

6. **Analyze the model:** After patterns have been identified, and the developer is satisfied with the initial results, three forms of analysis may be conducted on the model results: (a) ensure that all events are part of a template, (b) identify the likelihood of occurrence of each particular trace, and (b) derive an N-squared diagram or Design Structure Matrix (DSM) [160], [161], [167] of the results, so that all possible interactions can be shown in a single view-point and so that the results from additional runs at higher scope can be evaluated.

- (a) *All events in a template:* Universal templates can ensure that all events in all traces are part of a template, thereby ensuring the developer that there exist no sets of events that have not been evaluated. Execution at higher scope would be comprised of combinations of relatively few templates.
- (b) *Probability:* The probability of each sequence of events was derived by developing a post-execution script that employs Bayes theorem of conditional probability that is aligned to the events of the behavior analysis. Examples of Bayes theorem included those outlined by Duda [168], and were employed for verification of the approach. Significantly, the conditional probability construct needs to be fully consistent with the constraints employed by the model. The constraints fundamentally change the belief-network of the probability structure. Further, the constraints are a form of conditional probability, often written as an if-then statement.
- (c) *The DSM:* Since each trace is represented as a directed graph, without loops, deriving a DSM is straightforward. The directed graph consists of nodes, comprised of the events, and edges, comprised of all of the interactions among the events. Each edge, or line and arrow of the graph, begins at a source node (source event) and ends at a target node (target event). By tabulating each interaction in a matrix form for each trace, and then building this tabulation from trace to trace, the developer can ascertain a compilation of all interactions within the schema in one view. A script was developed in order to

automatically derive the DSM from an MP output file. Simple inspection of the DSM may include consideration of rarely employed interactions. If problematic behaviors exist, the developer may consider a of design constraint to eliminate that unwanted behavior. An algorithm was also developed to compare multiple runs at differing scopes of execution. More complex analysis includes a means to form groupings of events that have a high degree of interaction, termed a clustering analysis. Post-analysis employs a form of a clustering algorithm, developed by Thebeau [162]. This algorithm was applied to the DSM derived from MP, providing a means to systematically gain additional perspective of the interactions derived from the system architecture.

7. ***Model and Template Reuse:*** Reuse of the model and the associated templates has the potential to significantly reduce the labor associated since the models can be applied to a wide range of problems. Consider these reuse aspects separately.

- *Reuse of the Model:* An abstracted model may represent multiple applications by implementing an architecture that captures the relevant behavior events necessary by all applications, as illustrated in Figure 4.2. The examples outlined in Section 4.3.1 and 4.3.2 follow this approach. In the former example, the model was developed with a relatively high level of abstraction. For example, the leader is an abstract concept that applies to both the surgeon and pilot which is relatively easy for the reader to mentally keep track of the association. The latter example illustrates a level of difficulty in dealing with abstraction. The applications are (1) for a shooter to hit a target or (2) for a rescuer to save a victim. The generalization is that the actor to act upon an object. In this case, the abstracted layer becomes more difficult to interpret: “a hostile interpreted as friendly” is much easier for the developer to comprehend than “an object interpreted as a not an object of interest.” Yet, provided that the logic is consistent, the automated query will work in any case.
- *Reuse of the Template:* relies upon employing the template in a way that searches for event attributes that follow the semantics of a governing ontology, conceptually illustrated by Figures 4.3 and 4.4. The Observe, Orient, Decide, Act (OODA) loop segment template was used for each of the three applications from Sections 4.3.1 and 4.3.2. The concept of the Observe, Orient, Decide, Act (OODA) loop was developed by Boyd [35] and further documented by Coram [169].

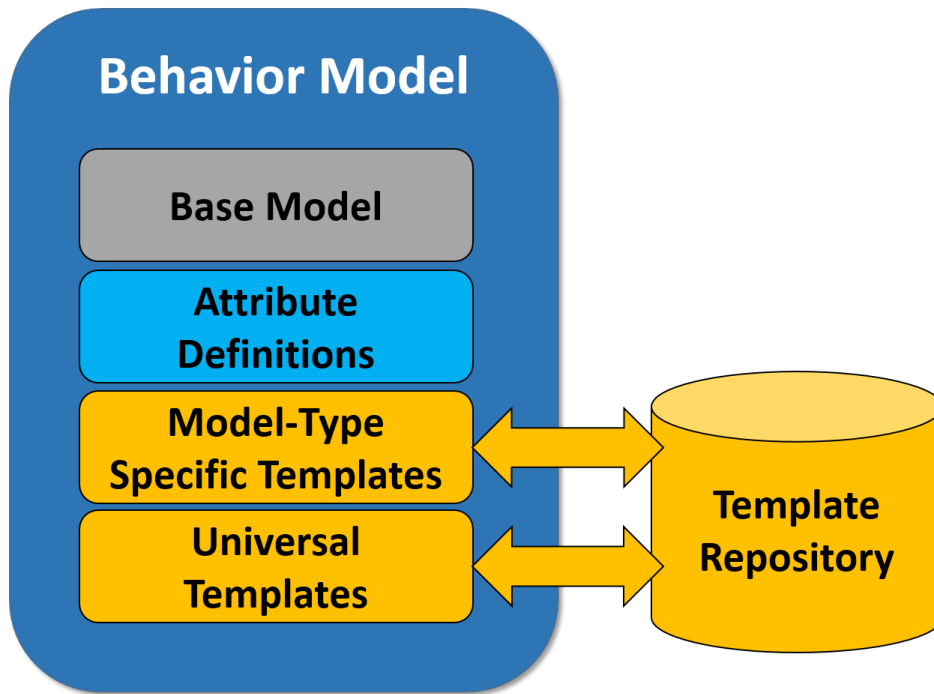


Figure 4.2. Reuse concept using a template based on attributes that follow semantics aligned with a governed ontology.

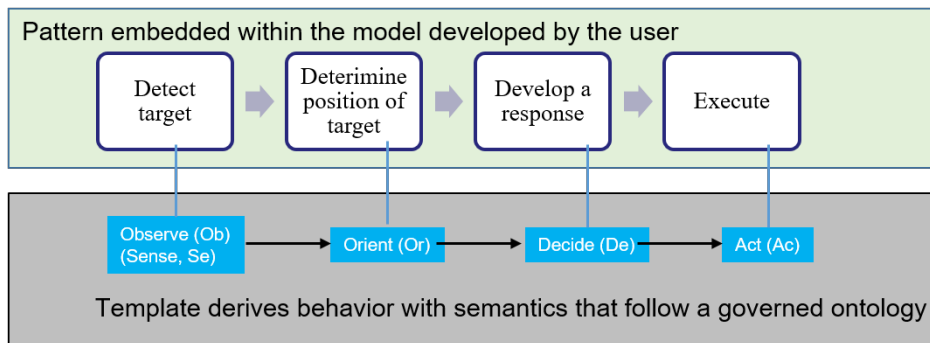


Figure 4.3. Observe, Orient, Decide, Act (OODA) loop, applied as a template in order to identify the associated inherent pattern of the system.

This section details several example problems that demonstrate the methodology and inform the findings of the dissertation. Three example problems were extensively investigated, including a failure mode problem that examines the interactions among a leader and subordinate(s), an SoS data exchange model that examines the hand-off of critical information to complete a mission, and

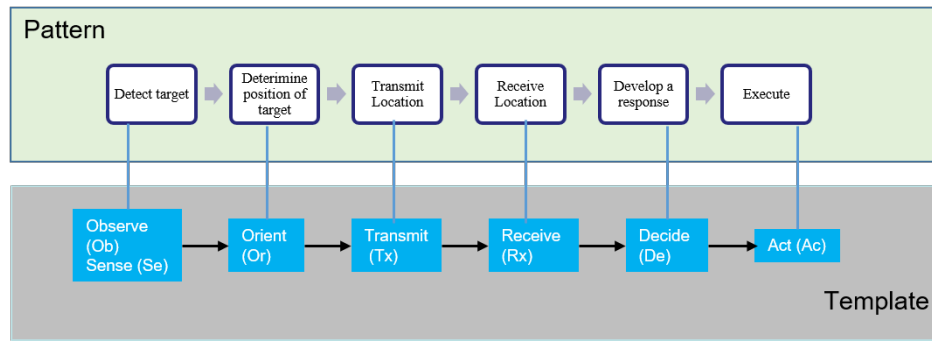


Figure 4.4. Coordinated OODA loop template, indicating the existence of a transfer and receive segment within the sequence.

a designed SoS model that includes layers of design from the component of participant systems to the combined system capability. These examples demonstrate the depth of analysis associated with the proposed methodology, illustrating the level of detail that can be accomplished even using models at a higher level of abstraction. These models also serve as a means of verification of the methodology.

4.1.1 Implementing Monterey Phoenix (MP) within the Methodology

MP uses pseudo-code, or schema, in order to instantiate particular behaviors within the model. Each section of code is written and edited in the designated text-box. The scope of execution controls the number of iterations computed in accordance with the schema. Upon execution, diagrams of an exhaustive list of traces are produced.

The MP execution environment is the available through MP-Firebird tool, illustrated in Figure 4.5. MP-Firebird supports the import of user-defined code as well as readily accessible examples. The environment also supports export of code and diagrams.

Monterey Phoenix (MP) applies the following fundamental construct, as outlined in the syntax manual [66]:

- the MP applies a light-weight formal method, implementing a formal language and a precise specification without incurring the cost of a formal proof [114].
- MP employs a powerful language, or pseudo-code, that formally describes the behaviors within the system. The complete model is referred to as a schema.
- All blocks in the system are considered as events.

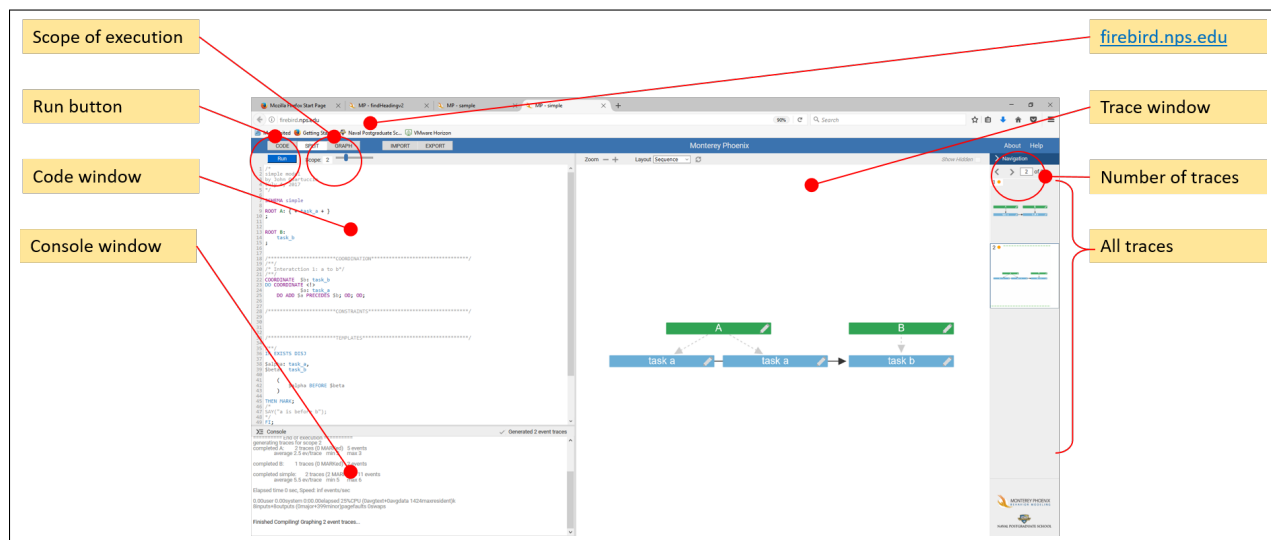


Figure 4.5. The Monterey Phoenix (MP) model environment, illustrating the code window, scope of execution, the current trace, and all traces produced by executing the schema.

- Root events initiate the hierarchy of subordinate events. The subordinate events are established by an *inclusion* relationship stemming from the root event.
- Events also exhibit *precedence* relationships both within a root hierarchy and coordinated among multiple root hierarchies.
- MP separates the definition of behaviors within the system from interaction with other systems, users, or the environment through coordination of inter-related events.
- Atomic events are comprised with no constituent event or attribute, such that these events have a parent event in the hierarchy, but no children.
- Composite events are comprised with one or more constituent events or attributes, such that these events have both a parent event in the hierarchy and at least one child event or attribute.
- Execution of MP exhaustively produces all possible outcomes in accordance with the schema and within the scope of execution.

Key point:

Execution of an MP schema produces all possible outcomes within the scope of execution, thereby enabling a means to identify all intrinsic behaviors of the model.

4.1.2 Methodology Description

Six steps of the proposed methodology to build behavior models that support System of Systems (SoS) development are shown in Figure 4.1. SME input is fundamental to successful implementation of the model, ensuring that critical aspects of the design are addressed in the model. At nearly all of the steps, the model is executed by the developer such that only small changes are implemented at a time. Then the effects are immediately analyzed within the MP-Firebird environment. This immediate feedback, produced in the execution environment, encourages interaction and engagement by the developer with the central engagement of a SME. Also at each step, the developer considers whether criteria are met to complete the step and then decides whether to proceed to the next step, repeat work in the current step, or go back to any previous step. The following sections describe the methodology as a step-by-step process to develop an MP model, one set of commands at a time.

4.1.2.1 Step 1: Develop a narrative of the behavior

It is typical that the model developer is not the end-consumer of the model and associated analysis, and so SME input is critical to the process. The SME may be an end-user of the product, the funding sponsor of the project, or perhaps the developer of a legacy or predecessor system. The model developer begins the methodology by interviewing the relevant SMEs necessary to formulate a complete, analyzable, internally consistent, and elegant description, in accordance with qualities of great models [170].

Criteria: This step is complete when the SMEs have described the relevant behaviors, constraints, assumptions, and limitations of the SoS. A written and approved document is the recommended practice, especially if significant resources are spent in order to develop the model.

Decision: Proceed to next step, repeat this step, or go back to previous step, as follows:

Proceed if documentation provides an internally consistent starting point, then proceed with the model development. Further input will be needed at the next steps, and so the description will may not be comprehensive for the first cycle of the methodology.

Repeat if the description is not consistent in its description and definitions; clarification will be necessary.

Go Back if the SME input cannot be reconciled, consideration needs to be given to address conflicts with sponsoring authority, as necessary.

Key point of the narrative:

Subject Matter Expert (SME) input is critical to modeling a system architecture.

4.1.2.2 Step 2: Identify MP events

A simple example is used throughout each of the next sections in order to illustrate how to describe events in MP, as indicated in Figure 4.6. This model includes two root events A and B, with corresponding atomic events “task_a” and “task_b.” Root events are typically the environment, a user, a system, or a component of the system. Subsequent events are typically actions taken by the root.

The inclusion relationship is represented with the dotted line. The model was executed with scope of two, producing two traces. Trace 1 has a single atomic event for both roots A and B, while trace 2 has two atomic events for root A and a single atomic event for root B.

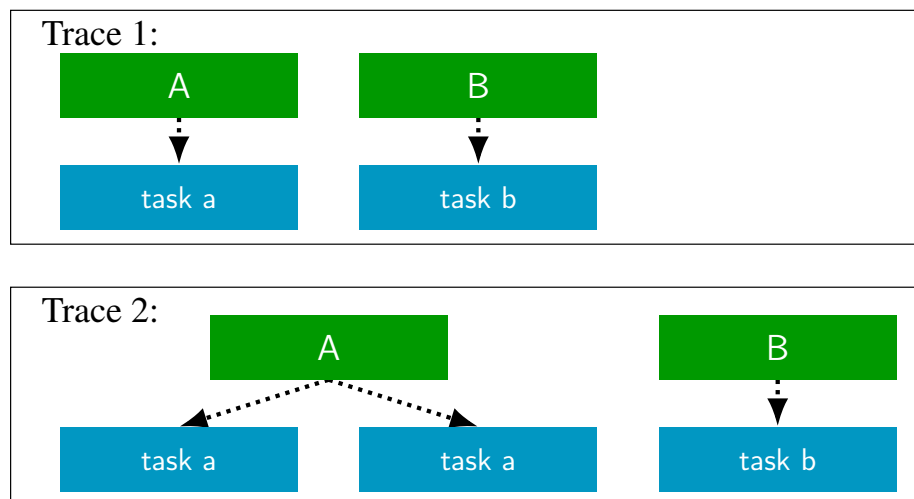


Figure 4.6. A simple Monterey Phoenix (MP) model illustrating root events A and B, with corresponding atomic events “task_a” and “task_b”. This inclusion relationship is represented with the dotted line. The model was executed with scope of two, producing two traces. Trace 1 has a single atomic event for both roots A and B, while trace 2 has two atomic events for root A and a single atomic event for root B.

The code for this model is listed below, where the curly brackets “{...}” indicate concurrent events, and the plus sign, “+,” surrounding task_a indicates that the number of events is at least one and up to the scope of execution. Since this schema was executed at a scope of two, the output includes

a trace with one “task_a” event and a second trace with two “task_a” events. Both traces have a single “task_b” event. Each root description ends with a semicolon, “;”.

MP code segment 4.1: An example of event definitions.

```
SCHEMA simple
ROOT A:
{ + task_a + };
ROOT B:
task_b    ;
```

Criteria: This step is complete when the relevant environment, users, systems, and components are defined as root events, and all functions are captured as atomic or composite events.

Decision: Proceed to next step, repeat this step, or go back to previous step, as follows:

Proceed when coordination with the SMEs affirm that all events are captured.

Repeat if the SMEs identify significant errors.

Go Back if fundamental problems exist in the narrative.

4.1.2.3 Step 3: Identify coordination

Coordination establishes precedence relationships between events within roots A and B. In the simple illustration we can establish that in all cases, “task_a” precedes “task_b”, as illustrated in Figure 4.7, with the code listed below the figure. The “DO...OD” command establishes a loop for asynchronous coordination so that all occurrences of “task_a” precede “task_b.” Events may also be shared by multiple roots using a “SHARE ALL” command.

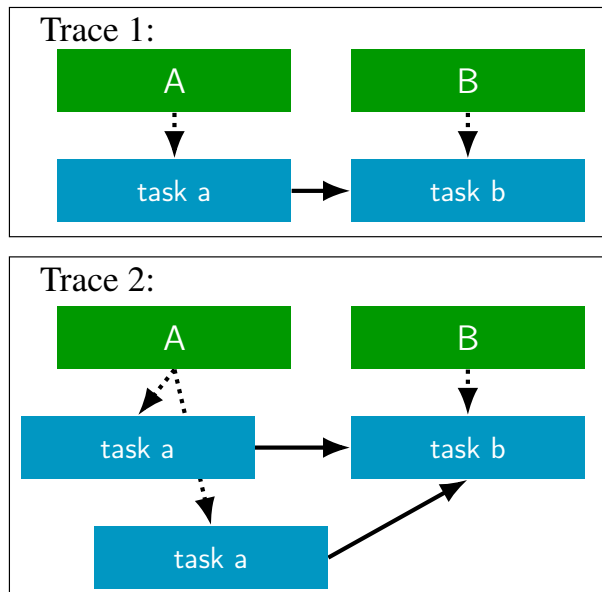


Figure 4.7. A simple Monterey Phoenix (MP) model with coordination between atomic events from root A and B. Coordination from task a to task b for both traces are illustrated as solid lines, representing precedence relationships. As such, task a must be accomplished before task b.

MP code segment 4.2: An example of adding a precedence relationship.

```
COORDINATE $b: task_b
DO COORDINATE
  $a: task_a
  DO ADD $a PRECEDES $b; OD; OD;
```

Criteria: This step is complete when coordination across all root events are complete.

Decision: Proceed to next step, repeat this step, or go back to previous step.

Proceed if the developer and SMEs agree with the coordination structure.

Repeat if errors are found.

Go Back if the root events need to be restructured.

Key point:

Monterey Phoenix (MP) separates the behavior definition within a system, modeled as a root event, from the interaction among the System of Systems (SoS), modeled as multiple root events.

4.1.2.4 Step 4: Define constraints

Four types of constraints are of concern to the developer. These include logical, simplification, design, and definition. These constraints are derived interactively by the developer and constitute a formal description of system requirements.

A ***logical-type*** of constraint inhibits certain behaviors in order to maintain a realizable representation of the system. For example, consider a situation where a book may exist or not exist, and a student shall either read or not read the book. It is impossible for the student to read a book that does not exist, and so the model developer must restrict this sequence from the model. This is an example of a logical-type of constraint to be implemented in the model.

The developer also needs to consider the impact of irregular activities within the model, especially in situations where emergent behaviors may exist, but are not obvious. Consider a situation where a passenger train can either stop at a station or not stop at a station; and a passenger at the station can either board the train or not board the train. The developer may decide to restrict the condition where the passenger boards the train if it has not stopped. Under most situations, this restriction is reasonable; however if searching for irregular interactions, the constraint will inhibit the behavior that is of interest to the developer. Irregular behavior may include the activity of a person illegally boarding a moving train. If restricted, the developer would not identify this potentially emergent event. Therefore, the developer needs to ensure that the model represents all areas of concern, by not overly-constraining the model.

A ***simplification-type*** of constraint is conducted at the discretion of the developer in order to concentrate the effort of the model where significant impact is to be expected. For example, if a student completes an assignment and turns it in to the professor, and the professor is responsible to grade all assignments that are turned in, a reasonable assumption is that the student's assignment is graded by the professor. This is a simplification-type of constraint that enables a comprehensive structure of the model, but it will limit the results to areas of greater interest to the developer.

A ***design-type*** of constraint establishes requirements of the system in order to ensure that sufficient

boundaries are placed on the interaction of relevant events. For example, if a developer was interested in finding a means to ensure that vehicle operators remain alert while driving their car, it would not benefit the developer to target children under twelve years old for the methods or intervention. Considering a physical system, if an aircraft is executing a landing mode, it would not make sense for the vehicle to simultaneously attempt to re-fuel from another aerial platform. This would be a typical design-constraint such that certain behaviors are limited to particular modes or states of the platform.

A **definition-type** of constraint may be used in order outline a particular sequence of events. For example, an environment including a problem may include a hazardous condition (e.g., aircraft with low fuel), and that condition would be defined as a particular event or sequence within the model.

Implementing constraints for our simple example: if the schema were executed without the coordination of precedence relationships, the developer would find the occurrence of a trace shown in Figure 4.8, where “task_b” occurs without a logical predecessor of “task_a”. A read function cannot occur before a write function, nor a receive function before a send function. Clearly in these cases, a logical-type of constraint is needed for the model. The following code ensures that the number of “trace_a” events is greater than the number of “trace_b” events.

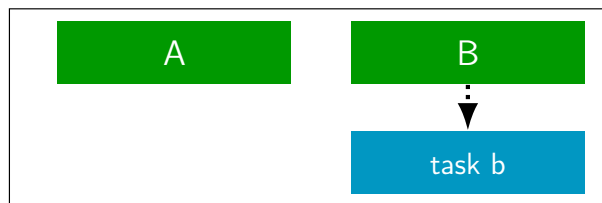


Figure 4.8. A simple Monterey Phoenix (MP) model with a missing constraint, allowing a trace in which “task_b” exists without a corresponding “task_a.”

MP code segment 4.3: An example constraint.

```
ENSURE ( #task_a >= #task_b );
```

Criteria: This step is complete when all logical, simplification, and design constraints are implemented in the schema.

Decision: Proceed to next step, repeat this step, or go back to previous step.

Proceed when the developer and SMEs agree that the constraints are an appropriate representation.

Repeat if constraints are missing or overly used, restricting desirable or actual behavior.

Go Back if any part of the event structure needs to be reworked in order to satisfy the needed behaviors.

Key points:

1. Constraints may be used to identify requirements that are necessary for enabling desirable behavior and restraining undesirable behavior.
2. Over-use of constraints may unintentionally hide intrinsic behaviors of the architecture necessary to identify inherent system behaviors that may reasonably be expected to occur.

4.1.2.5 Step 5: Identify patterns

Identifying patterns in the MP results, using the assertion checking functions along with the “MARK” and “SAY” commands has several purposes that are critical to the system developer including the following:

- Quickly investigate large sets of output that can easily reach in the tens of thousands of traces.
- Ensure that all traces contain only expected output.
- Develop a repository of effective architectures that can be employed in future designs, thereby capturing positive lessons learned.
- Develop a repository of ineffective or problematic architectures, including unwanted sequences, thereby capturing negative lessons learned and system warnings.

Figure 4.9 illustrates how the assertion checking function is used in MP. The code listed below the figure checks for conditions where “task_a” occurs before “task_b.” If the condition is met, the trace is marked and annotated with the “SAY” command. Each of these checks can be formulated as templates of unique behavior in the model. Long term objectives include aligning a repository of these templates to enable re-use of working and validated architectures. As importantly, the repository would also include abandoned, ineffective, or problematic architectures and implemented as warnings to the developer.

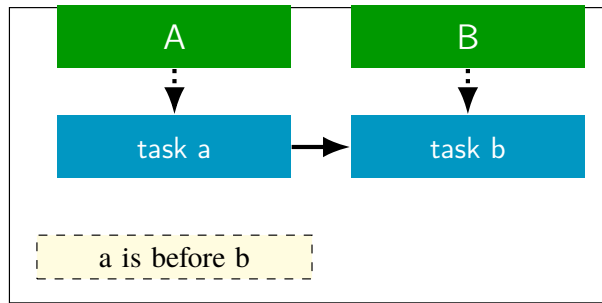


Figure 4.9. A simple Monterey Phoenix (MP) model with assertion checking such that each trace that satisfies the condition that “a is before b” is marked and labeled. This capability establishes the means to identify templates, or patterns, in the results.

MP code segment 4.4: An example pattern.

```

IF EXISTS
  $alpha: task_a,
  $beta:  task_b
  (
    $alpha BEFORE $beta
  )
THEN MARK; SAY("a is before b");
FI;

```

Criteria: This step is complete when templates are associated with desirable and undesirable behaviors of the model.

Decision: Proceed to next step, repeat this step, or go back to previous step.

Proceed when the developer and SMEs agree that the templates have been effectively implemented.

Repeat if significant portions of the output cannot be resolved as expected behavior.

Go Back if additional constraints are needed in the model to avoid unwanted patterns.

Key point:

Identifying patterns of behavior assists the developer to find positive and negative sequences within the architecture.

4.1.2.6 Step 6: Evaluate the results

Several types of analyses may be of interest to the developer, using the results from executing the MP model, including the following:

- Probability of any trace or outcome
- A summary of all interactions in the form of a Design Structure Matrix (DSM) or N-squared Diagram
- The throughput of data flow
- The sequencing and timing of events
- Requirements on computing power

For the purpose of this chapter, just the first two types of analyses will be explored as follows:

Analysis of probability of each trace may be straight-forward for a very simple model, but the developer needs to ensure that the fundamental principles of statistics are followed closely. This section outlines certain conditions where an MP model can be used to define a Bayesian belief-network, and then probabilities of each trace could be derived from a given set of assumptions [5]. The conditions include the following:

1. Each trace is an instance of the architecture and is represented as a directed graph without loops. This is a given for MP.
2. Each trace has the same topology, such that a consistent Bayesian belief network may be applied to all traces. This is not typical for many models, since a change in scope will change the topology, reflecting additional iterations. And so the developer needs to maintain a single scope for all traces in the output.
3. Constraints need to be written as conditional probabilities in order to be integrated into the belief network.
4. An approach is defined to prorate probability after multiple constraints are applied to the model. Constraints tend to eliminate large segments of the belief network.
5. Of note, the impact of an overly-constrained model will become apparent when attempting to implement a prorating scheme.

Key point:

Probability calculations may uncover overly-constrained models that inhibit realistic modes of behavior.

A Design Structure Matrix (DSM) can be derived directly from the MP model, as indicated in Table 4.1. The DSM, also called an N-squared diagram, forms a matrix by taking every event along the row and column, and then identifying the interaction of events within the matrix. As outlined

by Browning [160] and Eppinger and Browning [161], DSMs have been used to identify spatial requirements, energy flow, information flow, and material flow across systems. These are helpful to the developer because they support an additional form of review, particularly when the developer may not be aware of a particular behavior in the system. Consider the following activities:

1. Inspection of rare occurrences of interactions between events. These may have slipped the notice of the developer, especially with large data sets.
2. Investigation of common occurrences of desirable behaviors in the model,
3. Comparison of results with different scopes of execution, ensuring that no unexpected events occur.
4. Re-ordering of events so that clusters of common exchanges can be centralized and support system level instantiation during development [160]–[162].

Table 4.1. A simple Design Structure Matrix (DSM) derived from the MP model.

FROM	TO	A	task_a	B	task_b
		1	2	3	4
A	1	0	3	0	0
task_a	2	0	0	0	3
B	3	0	0	0	2
task_b	4	0	0	0	0

Key points about the DSM:

1. A DSM may be employed to identify and investigate rarely exercised interactions in the model.
2. Comparison of the DSM from multiple runs at different scopes of execution provides a means to indicate whether new behavior patterns exist.

Criteria: This step is complete when evaluation criteria is calculated or analyzed (e.g., probability of each trace, probability of success, effect of scope on execution, or other parameter).

Decision: Proceed to next step, repeat this step, or go back to previous step.

Proceed in order add detail or complexity to the behavior narrative for another full cycle of the methodology.

Repeat if added detail is needed in the analysis, such as executing the existing model at higher

scope in order to identify any unique occurrences in the data.

Go Back if the events, coordination, or constraints need to be revised.

When is the model complete?

The model is complete when requirements are identified in terms of constraints and the model has provided sufficient detail to formulate a simulation of system behavior. Types of models that form the basis of simulation include behavior models, physics-based models, agent-based models, discrete event models, or a combination of these methods.

4.1.3 Model Re-use

The simplest form of model re-use is a word-for-word replacement from one application to the next. However, the developer community may be better served by developing an abstracted model that may have clearer relevance to applications outside of the normal domain of research. Consider the cycle ISP model derived by Giammarco and Troncale [34], and depicted in Figure 4.10.

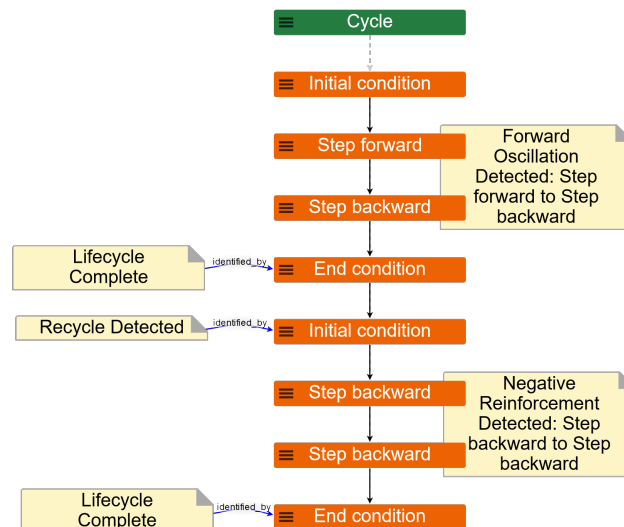


Figure 4.10. Cycle ISP model, executed at scope 2, trace 8. Adapted from [34].

The following code lists the commands necessary to produce two cycles of the desired behavior; however, the developer may wish to develop additional applications that follow the same form, without recreating the entire schema, including all associated templates.

MP code segment 4.5: Cycle ISP abstraction-level MP code.

```

ROOT Cycle: ( + <2> Initial_condition
              ( + ( Step_forward | Step_backward ) + )
              End_condition + );

```

Figure 4.11 illustrates a means to extract the attributes of the basic cycle model, leaving the topology and semantics as a means to identify the associated templates. These precise templates, if preserved in a repository, enable future developers the ability to readily exploit previous work by simply applying these attributes to their new model, while retaining the identical search patterns.

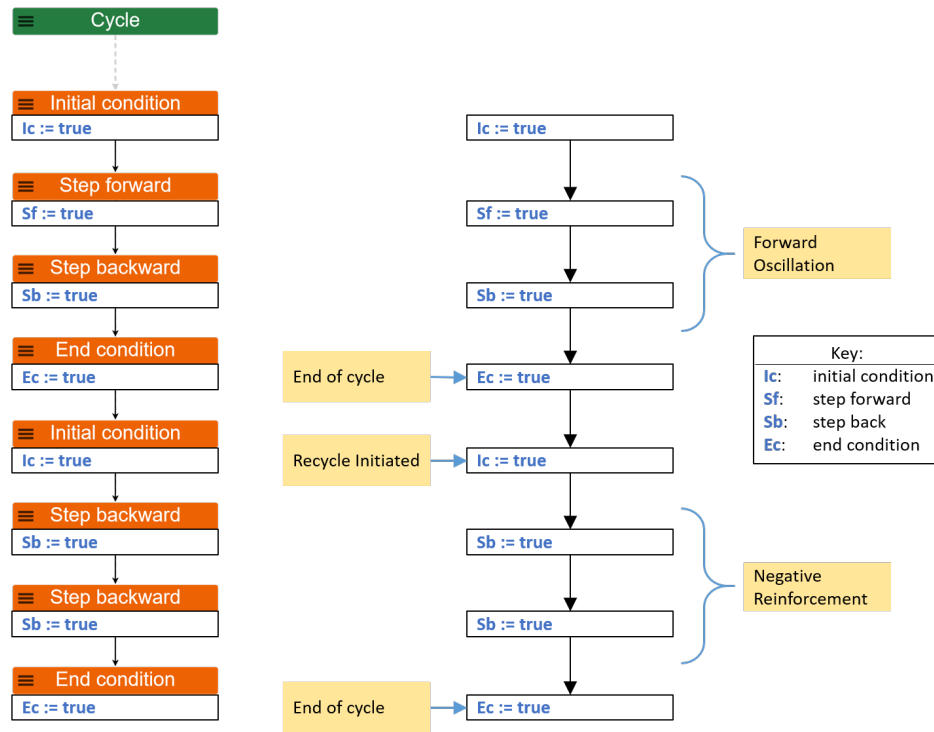


Figure 4.11. Cycle ISP model attributes identified.

MP code segment 4.6: Attributes aligned to the abstraction-level model.

```

ATTRIBUTES {bool Ic, Ec, Sf, Sb ;};
Initial_condition : BUILD{Ic:=true;};
End_condition    : BUILD{Ec:=true;};
Step_forward     : BUILD{Sf:=true;};
Step_backward    : BUILD{Sb:=true;};

```

MP code segment 4.7: Example Cycle ISP template segments.

```

/*Positive Reinforcement Template */
COORDINATE      $a:$$EVENT SUCH THAT $a.Sf
DO COORDINATE   $b:$$EVENT SUCH THAT ($b.Sf AND $a PRECEDES $b)
DO COORDINATE   $c:$$EVENT SUCH THAT ($c.Ec AND $b PRECEDES $c)
DO SAY("Positive Reinforcement Detected: " $a " to " $b);
$c.pos :=true;
$a.inTemplate := true; $b.inTemplate := true;
$c.inTemplate := true; OD; OD; OD;

/*Negative Reinforcement Template */
COORDINATE      $a:$$EVENT SUCH THAT $a.Sb
DO COORDINATE   $b:$$EVENT SUCH THAT ($b.Sb AND $a PRECEDES $b)
DO COORDINATE   $c:$$EVENT SUCH THAT ($c.Ec AND $b PRECEDES $c)
DO SAY("Negative Reinforcement Detected: " $a " to " $b);
$c.neg := true;
$a.inTemplate := true; $b.inTemplate := true;
$c.inTemplate := true; OD; OD; OD;

/*Oscillation Template */
COORDINATE      $a:$$EVENT SUCH THAT $a.Sf
DO COORDINATE   $b:$$EVENT SUCH THAT ($b.Sb AND $a PRECEDES $b)
DO COORDINATE   $c:$$EVENT SUCH THAT ($c.Ec AND $b PRECEDES $c)
DO SAY("Forward Oscillation Detected: " $a " to " $b);
$c.osc := true;
$a.inTemplate := true; $b.inTemplate := true;
$c.inTemplate := true; OD; OD; OD;

/*Lifecycle Completion */
COORDINATE      $a:$$EVENT SUCH THAT $a.Ec
DO ADD SAY("Lifecycle Complete ") identified_by $a;
$a.inTemplate := true; OD;

/*Recycle Template */
COORDINATE      $a:$$EVENT SUCH THAT $a.Ic
DO COORDINATE   $b:$$EVENT SUCH THAT ($b.Ec AND $b PRECEDES $a)
DO ADD SAY("Recycle Detected") identified_by $a;
$a.inTemplate := true; $b.inTemplate := true; OD; OD;

```

At this point, with the templates operating on the attributes and topology of the model, the developer can easily re-use the templates directly from the repository and simply align the new model to the attributes. Consider the following code for the human rest period, followed by Figure 4.12.

MP code segment 4.8: Cycle ISP applied to the human rest period.

```

/***** Human Model *****/
ROOT Human: ( + <2> Initial_condition
              ( + ( Awake | Sleep ) + )
              End_condition + );
ATTRIBUTES {bool Ic, Ec, Sf, Sb };
Initial_condition : BUILD{Ic:=true;};
End_condition    : BUILD{Ec:=true;};
Awake             : BUILD{Sf:=true;};
Sleep            : BUILD{Sb:=true;};

```

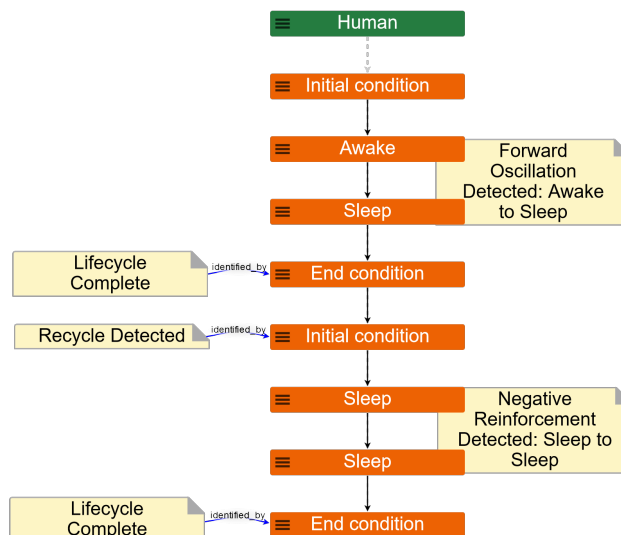


Figure 4.12. Cycle of human rest period, executed at scope 2, trace 8, employing the templates defined by attributes and topology.

Again, the following illustrates another application of a rock expanding and contracting as depicted in the code and Figure 4.13.

MP code segment 4.9: Cycle ISP applied to the expansion and contraction.

```

/*****Rock Model*****/
ROOT Rock: (+ <2> Initial_condition
            (+ ( Expand | Contract ) + )
              End_condition +);
ATTRIBUTES {bool Ic, Ec, Sf, Sb };
Initial_condition : BUILD{Ic:=true;};
End_condition     : BUILD{Ec:=true;};
Expand            : BUILD{Sf:=true;};
Contract          : BUILD{Sb:=true;};

```

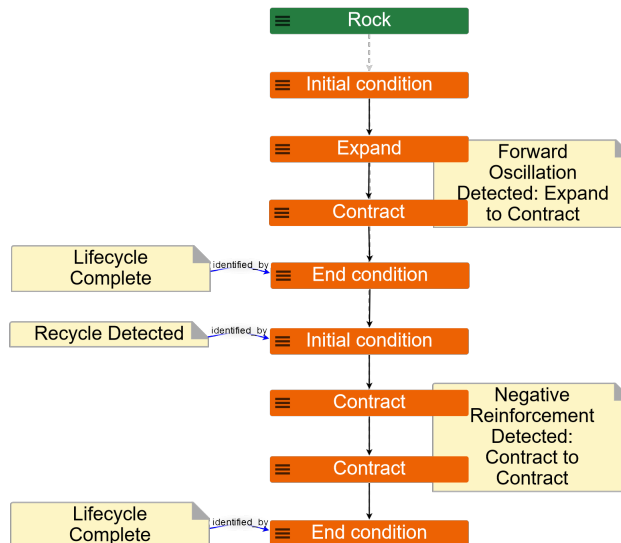


Figure 4.13. Cycle of expansion and contraction, executed at scope 2, trace 8, using the templates defined by applying the attributes and topology.

4.2 Derivation of Post-execution Analysis

4.2.1 Probability of Each Trace

This section illustrates the application of stochastic properties to Monterey Phoenix (MP) behavior models, given commonly used constructs. Each of these traces produced by MP are represented as directed graphs, without loops, thereby meeting the criteria as a Directed Acyclic Graph (DAG) [171]–[173]. The approach considers a means to align the probability belief network to the behavior model structures, including coordination, constraints, and conditional probabilities.

4.2.1.1 Fundamental axioms of statistics

The fundamental axioms of statistics govern the probability approach applied to MP. Consideration of the set of the sample space is of utmost importance. First, within this sample space, the probability of any included event is between zero and one, inclusive. Second, the sum of all probabilities within the sample space is equal to one. Consider the following derivation derived from the author's earlier publication [5]:

The proposed probability modeling process relies upon fundamental axioms [173]–[176] outlined below, Given a set of events, A , within a sample space, S , with discrete outcomes, N , the probability of event, A_i , is a real number between zero and one, inclusive.

$$0 \leq P(A_i) \leq 1 \quad (4.1)$$

The probability of the sample space, S , is one.

$$P(S) = 1 \quad (4.2)$$

Considering a second set of events, B , and if A and B are mutually exclusive, the union of their probabilities are additive.

$$P(A \cup B) = P(A) + P(B) \quad (4.3)$$

Furthermore, considering conditional probabilities [168], [172], [173], the probability of the intersection of A and B is the probability of B given A times the probability of A , which is also equal to the probability of A given B times the probability of B . This

relationship is the *fundamental rule for probability calculus*.

$$\begin{aligned} P(A \cap B) &= P(A, B) = P(B|A)P(A) \\ &= P(A|B)P(B) \end{aligned} \tag{4.4}$$

Bayes Theorem can be derived from the fundamental rule as follows:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \tag{4.5}$$

4.2.1.2 Probability structures

Within MP, the developer adjusts the model through coordination of the events and adding constraints that change the model structure, rejecting the associated traces as invalid. As most models are refined, user may expect that more traces become rejected than those that are accepted.

Consider the following aspects of applying probability structures to a behavior model:

1. *Domains*: Axioms of statistics apply to each of the following domains, where each domain creates a sample space. For each domain, the fundamental axioms of statistics must be maintained. As illustrated in Figures 4.14 and 4.15, applicable domains include the following:
 - (a) Iterations within the schema,
 - (b) Traces within each iteration,
 - (c) Traces within the schema,
 - (d) Alternatives of each event.
2. *Valid traces*: The probability aligned to valid traces within the schema must be derived consistently with the coordination and constraints of the model.
3. *Bayesian belief network*: Probability of events within a particular trace need to be adjusted conditionally on the basis of previous events within the trace.
4. *Prorating scheme*: Compounding effects of multiple constraints changes the structure of the probability belief network, and so some prorating scheme must be implemented.

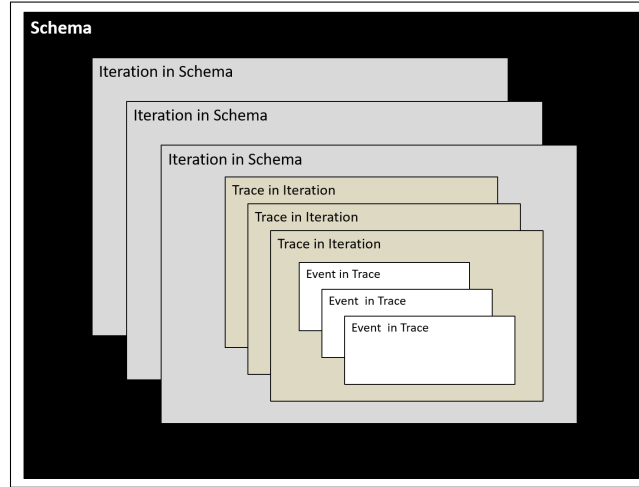


Figure 4.14. Domains within MP: Iterations within the schema, based on the scope of execution, Traces within each iteration, Total traces within the schema, and Alternatives of each event.

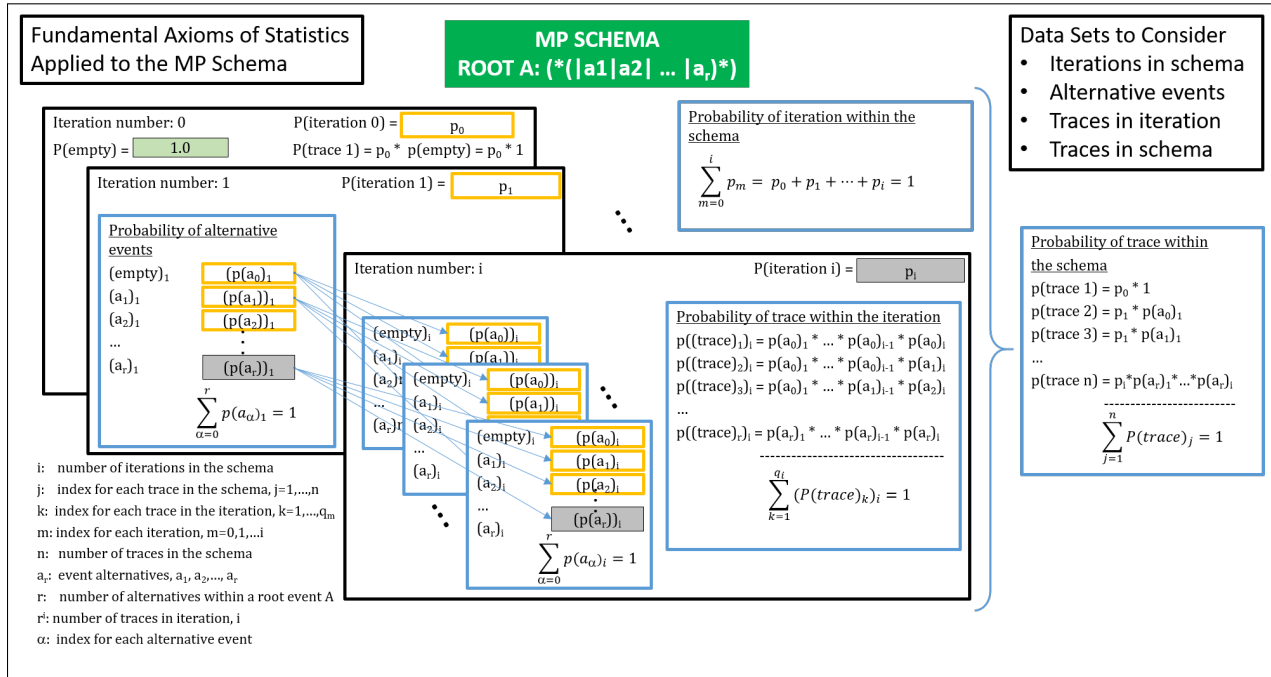


Figure 4.15. Fundamental probability axioms applied to the Monterey Phoenix schema. Sample spaces outlined in the blue boxes include (1) the iterations within the schema, (2) the traces within each iteration, (3) the traces within the schema, and (4) each set of alternative events. The values in the gold boxes may be user-defined. The value in the gray box is derived such that the sum of each sample space is equal to one.

4.2.1.3 Derivation of probability structures

This section describes how conditional probabilities can be implemented within MP, and then illustrates a process to align constraints to the associated Bayesian belief network. Finally, the section outlines an automated script, written in MATLAB[®] used to derive the probabilities associated with a constrained behavior model.

4.2.1.3.1 Conditional probabilities: As demonstrated by Duda [168, p.58], conditional probabilities may be applied to a belief network, as indicated in Figure 4.16. Figure 4.17 applies this example using basic functions in MP.

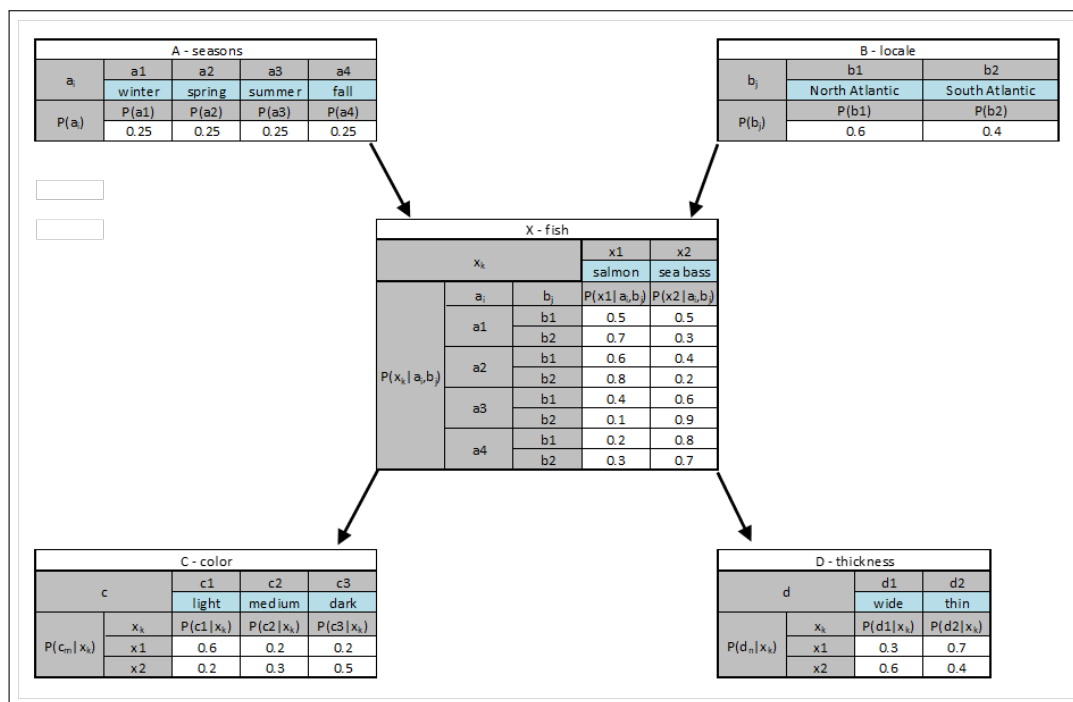


Figure 4.16. Conditional probability belief network. Adapted from [168, p.58].

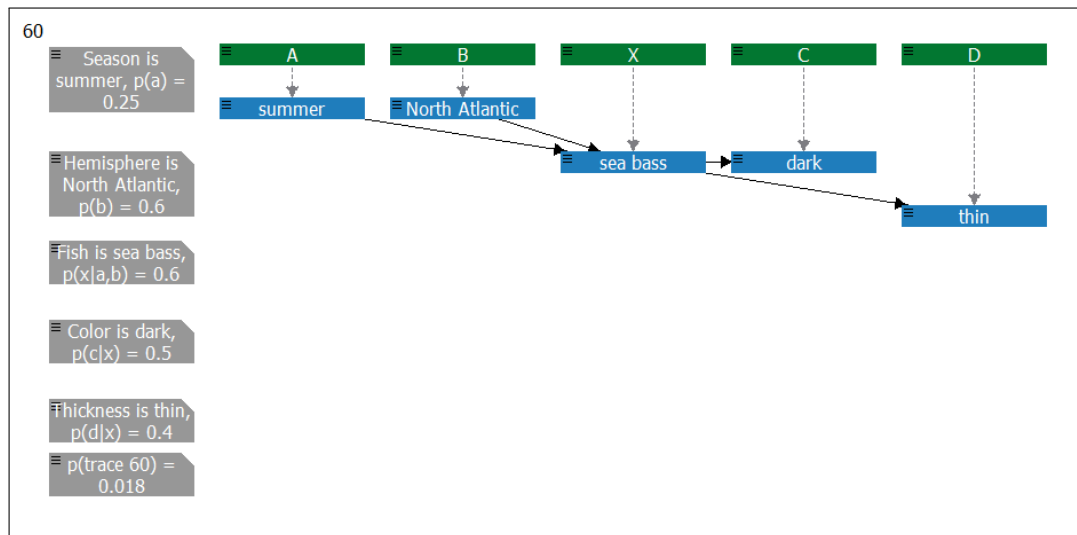


Figure 4.17. Verification of conditional probability calculation, illustrating trace # 60, with the same alternatives. Adapted from [168, p. 59].

4.2.1.3.2 Constraints aligned with the probability belief network: This section outlines a simple case illustrating how constraints need to be aligned with the system probability structure, followed by a situation where multiple constraints combine to change the belief network. This section was also derived from the author's previous publication [5].

A case with only a single constraint: Consider the following simple case that illustrates the importance of aligning the probability belief network to the constraints of a model. Assume that there is a probability of a car existing and a secondary probability that I would like to drive that car, as follows:

$p(\text{exists})$	$= 0.75$
$p(\text{not\_exist})$	$= 0.25$
$p(\text{drive})$	$= 0.6$
$p(\text{not\_drive})$	$= 0.4$

Question: What is the probability of me not driving the car?

An MP model representing this condition consists of two root events as follows:

```

SCHEMA driveCar
ROOT car: (exists | not_exist);
ROOT me: (drive | not_drive);

```

The following precedence relationship links the alternate events:

```

COORDINATE $e: (exists | not_exist),
            $d: (drive | not_drive)
DO ADD $e PRECEDES $d; OD;

```

Since it is not possible for me to drive a car that does not exist, I need a constraint as follows:

```

ENSURE(#not_exist == 1 -> #drive == 0);

```

The constraint is just a special case of conditional probability, such that the probability of driving a car that does not exist is zero, as follows:

$$P(drive|car\_not\_exist) = 0 \quad (4.6)$$

In order to maintain the second axiom listed in equation 4.2, the probability of not driving a car that does not exist is one, written as follows:

$$P(not\_drive|car\_not\_exist) = 1 \quad (4.7)$$

Answer: Using equations 4.3 and 4.4, we find that the probability of not driving a car may be represented as follows:

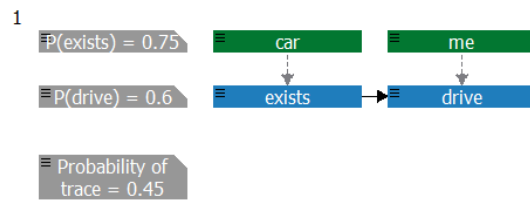
$$\begin{aligned}
P(not\_drive) &= P(not\_drive|car\_exists)P(car\_exists)+ \\
&\quad +P(not\_drive|car\_not\_exist)P(car\_not\_exist) \\
&= P(trace2) + P(trace3) \\
&= 1/2 + 1/4 = 3/4
\end{aligned}
\tag{4.8}$$

Figure 4.18 illustrates each of the traces. The code implements the probability values as attributes to each event, and aligns the conditional probability to be consistent with the constraints, as follows:

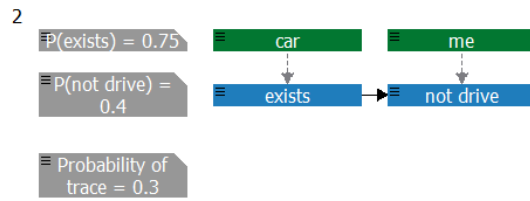
```

IF $e IS exists THEN $e.p:=0.75;      /*exists */
    IF $d IS drive THEN $d.p:=0.60; /*drive */
    ELSE $d.p:=0.40; /*not drive*/
    FI;
ELSE $e.p:=0.25;                      /*not exist*/
    IF $d IS drive THEN $d.p:=0;      /*drive */
    ELSE $d.p:=1; /*not drive*/
    FI;
FI;
p*:=$e.p;                             p*:=$d.p; /*$*/

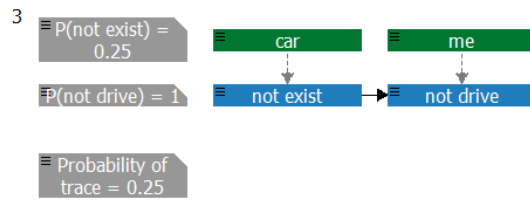
```



(a) Trace 1



(b) Trace 2



(c) Trace 3

Figure 4.18. Drive car model, demonstrating a constraint aligned with the probability belief network. Note that the sum of the probability of all traces in the schema are equal to one.

Multiple and compounding constraints: This condition frequently occurs since behavior models need to constrain behavior on the basis of logic, definition, design, or simplification. The following means was applied to derive a probability belief network that is consistent with the associated constraints on the model.

1. Populate the probability belief network without constraints, as illustrated in Figure 4.19.
2. Add conditional constraints that are consistent with the model, such that $p = 0$ for alternatives that satisfy the constraint, illustrated in Figures 4.20 and 4.21.
3. Prorate the remaining structure of the belief network such that the total probability of each sample space is equal to one, illustrated in Figure 4.22.
4. *Optional:* Display the prorated probability of each trace as MP output as illustrated in Figure 4.23

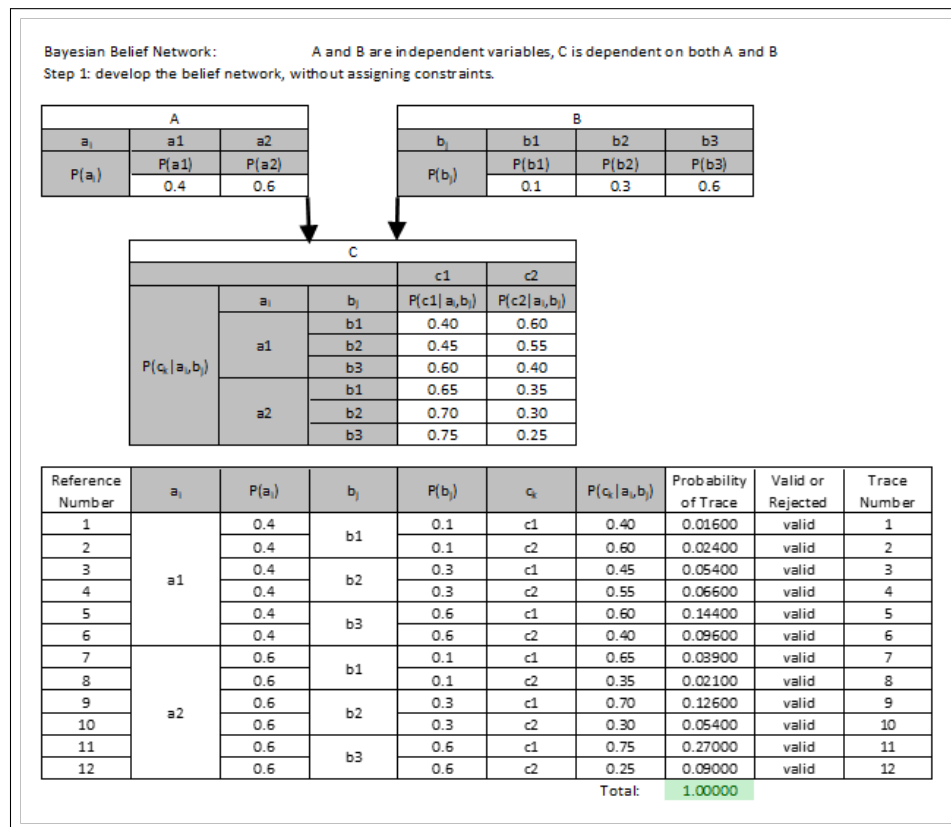


Figure 4.19. Conditional probability structure of a generalized behavior relationship where A and B are independent variables, and C is dependent on both A and B.

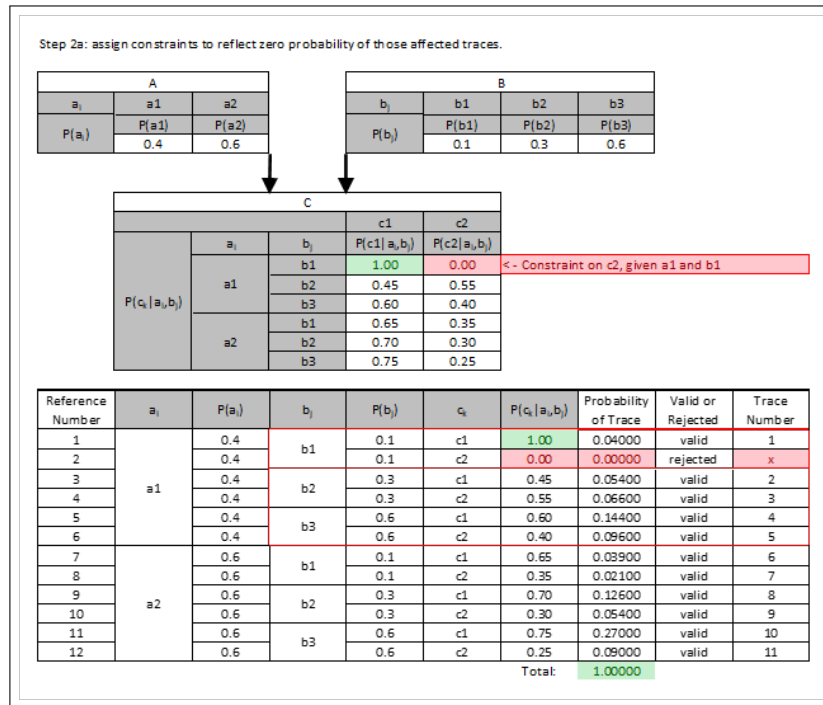


Figure 4.20. A single constraint added to the belief network, represented as a conditional probability, where the constrained behavior has a probability of zero and the alternate behavior has a probability of one.

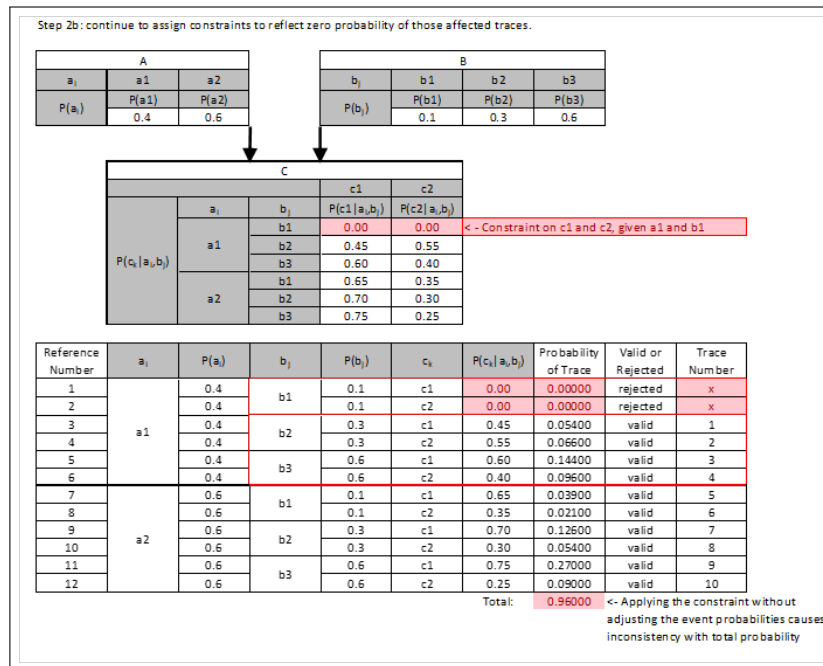


Figure 4.21. Multiple constraints along a single path causes the elimination of a corresponding segment. The probability assignments of a1 need to be adjusted such that $p(b_2 | a_1) + p(b_3 | a_1) = 1$.

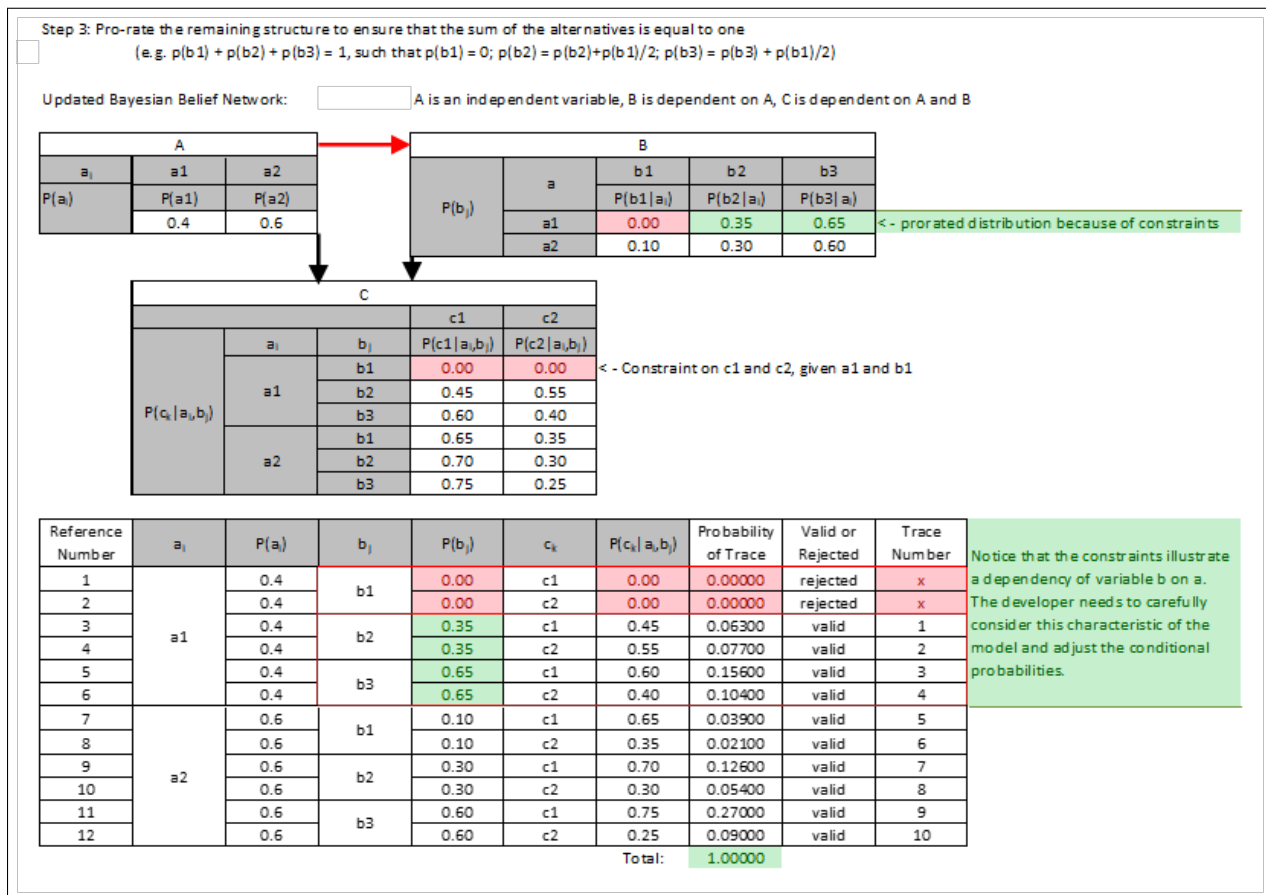


Figure 4.22. Prorating the belief network ensures that the sum of each set of alternatives is equal to one, thereby accounting for all traces and adjusting for the fundamental change in the belief network.

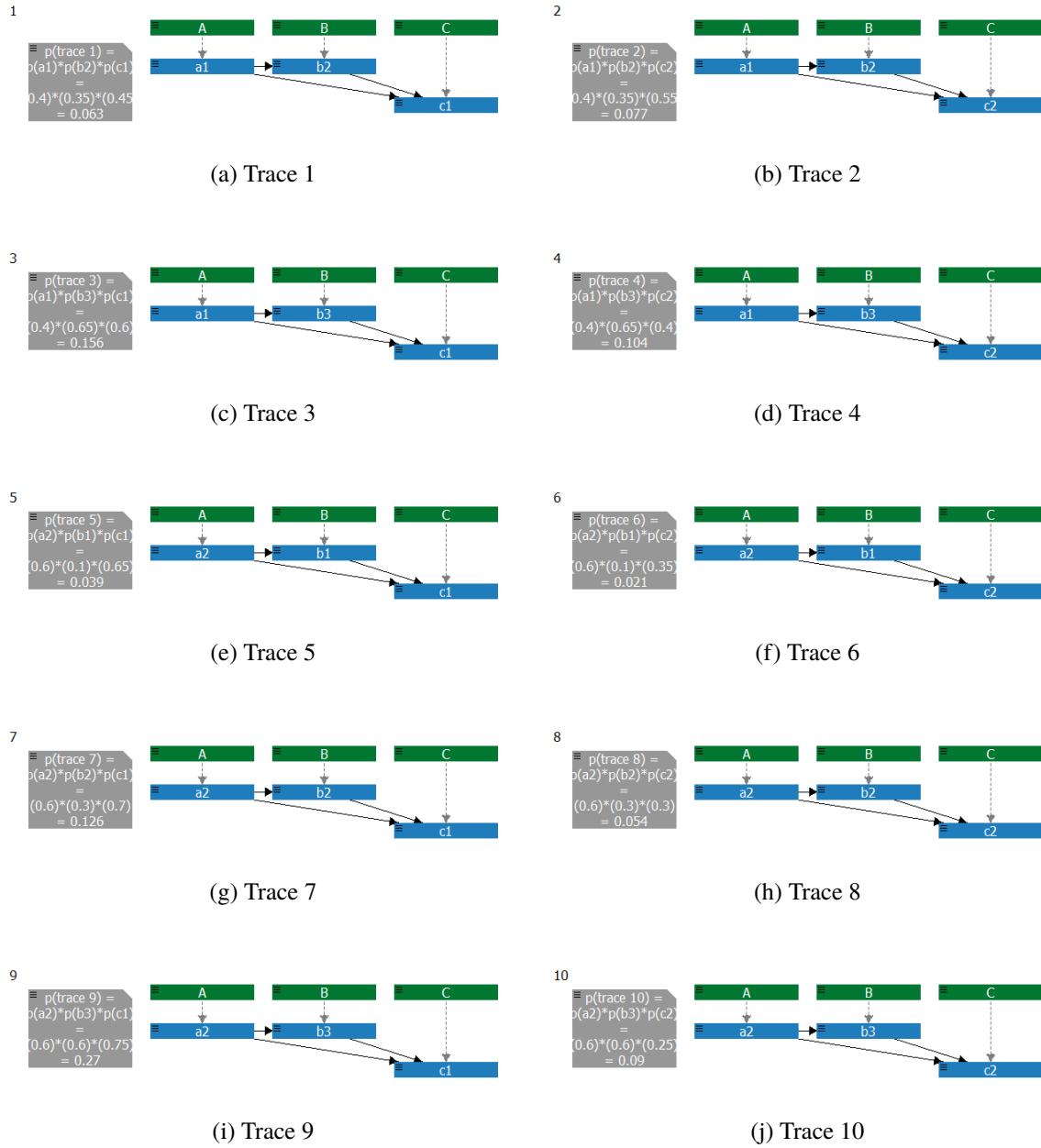


Figure 4.23. Constrained Bayesian Network.

4.2.1.3.3 An automated constrained probability script: This MATLAB[®] script aligns the behavior model constraints with the Bayesian belief network, such that the sum of probabilities for all of the traces in the schema is equal to one, and the sum of all alternative events at each node also equals one.

As illustrated in Figure 4.22, the Bayesian belief network consists of nodes A, B, and C. Node A is an independent variable, with two alternatives, indexed as $i = 1..2$ with probability of $P(a_i)$. Node B is dependent upon A, and has three alternatives, indexed as $j = 1..3$, such that $P(b_j|a_i)$. Finally, node C is dependent upon both nodes A and B, with two alternatives, indexed as $k = 1..2$, such that $P(c_k|a_i, b_j)$.

Appendix B lists the results from the MATLAB[®] script for the example problem illustrated in Figures 4.21 through 4.22. The core element of the script is the “trimBranch.m” function that uses the constrained belief network and then derives the prorating necessary to align to the constrained behavior model. Conditions for using this function include the following:

- The matrix representing the belief network be aligned from the left to right, meaning that the independent nodes are listed to the left of the matrix.
- All traces must follow the same topology, described as edges and nodes for a network, lines and boxes for an activity diagram, or events and precedence relationships for an MP behavior model.
- For more complex models, a particular event alternative could change subsequent topology, the developer may employ the function multiple times, as applicable to the “trimBranch” function.
- The developer must ensure that the constraints applied to the probability analysis using MATLAB[®] produce the same results as those derived from MP.
- The developer must investigate whether the results adjust the probability of one of the independent nodes in the belief network. In such a case, the model will either be overly-constrained, or the set of constraints are valid such that what was initially perceived as an independent node, now determined to depend upon an alternative from an earlier node.

Algorithm to derive model probability: A MATLAB[®] function, *trimBranch.m* was developed in order to derive the probability belief network of a constrained behavior model. As illustrated in Figure 4.24, the following steps are conducted by the associated script that was demonstrated in Appendix B:

1. Derive all possible traces, given the set of alternatives for each event,
2. Populate a Bayesian belief network with an initial set of probability of each event,
3. Apply constraints to the belief network, written as conditional probabilities,
4. Prorate the belief network, using the *trimBranch.m* function,

5. Calculate the probability of each trace using the prorated belief network.

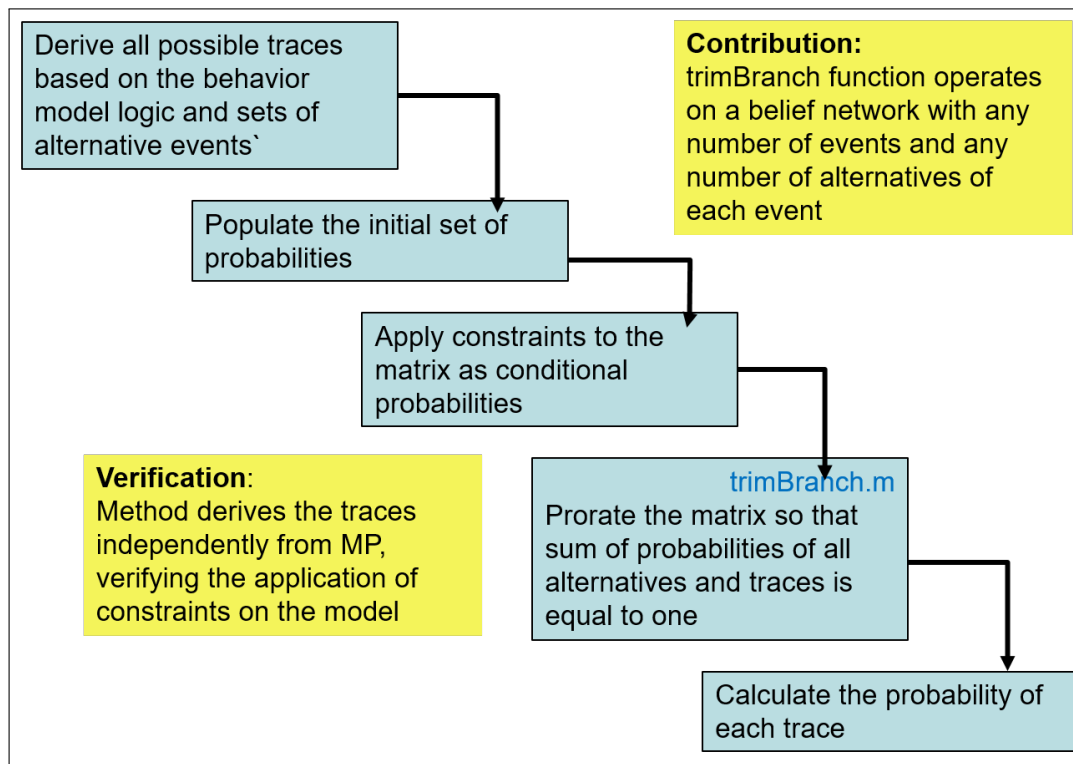


Figure 4.24. Prorating the constrained probability network by implementing the *trimBranch.m* function.

4.2.2 The Design Structure Matrix (DSM)

The Design Structure Matrix (DSM) is a form of an N-Squared (N²) diagram, that tabulates the directed relationships of a model between the rows and columns of a matrix. This matrix may be formed as Inputs in Rows (IR) where the input is from the row to the column, or as Inputs in Columns (IC) where the input is from the column to the row. When employed as an analysis method for behavior models and MP results, the developer has a means to review all interactions among all events in a single diagram. Figure 4.25 provides a simple example of an MP model, while Figure 4.26 illustrates the associated derivation of the DSM.

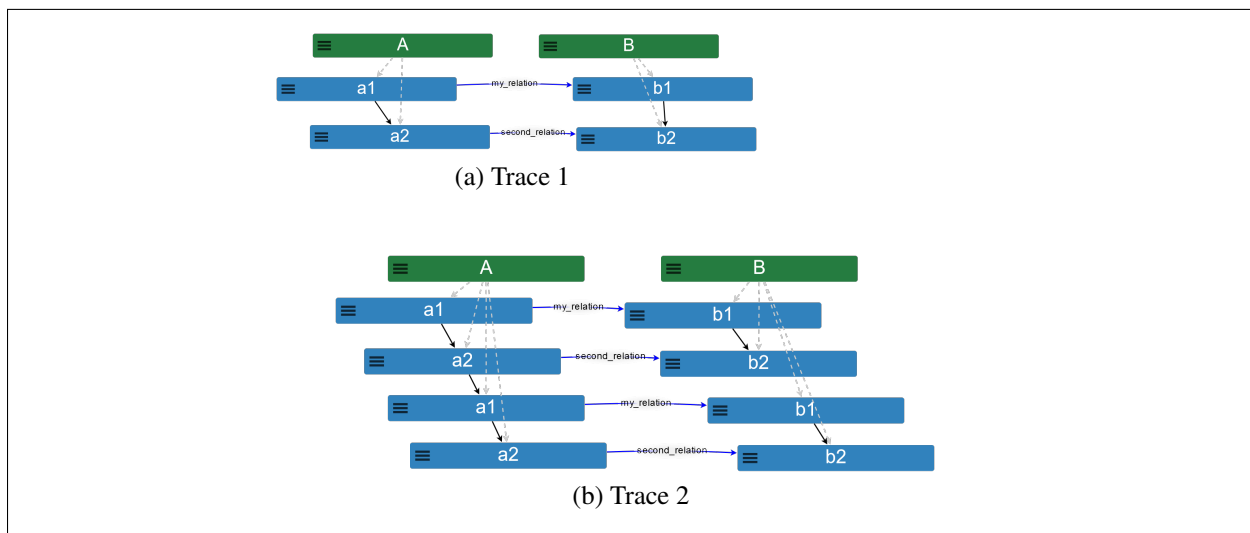


Figure 4.25. Example traces, illustrating user-defined relations from event a1 to b1 and a2 to b2, respectively. Adapted from [177].

MP code segment 4.10: Example 3. Source: [177].

```
SCHEMA e3
ROOT A: (* a1 a2 *);
ROOT B: {+ b1 b2 +};

COORDINATE $a: a1, $b: b1
DO ADD $a my_relation $b ; OD;

COORDINATE $a: a2, $b: b2
DO ADD $a second_relation $b ; OD;
IF #a1 > 1 THEN MARK;FI;
```

Event type	Event name		A	B	a1	a2	b1	b2
		index	(1)	(2)	(3)	(4)	(5)	(6)
ROOT	A	(1)			3	3		
ROOT	B	(2)					3	3
ATOM	a1	(3)				3	3	
ATOM	a2	(4)			1			3
ATOM	b1	(5)						3
ATOM	b2	(6)						

(a) The DSM of the example schema, shown in Inputs in Rows (IR) form, such that the source event is along the row and the target event is along the column. This view shows the number of interactions on the diagram with the lowest values shown in red and the highest value shown in green.

Event type	Event name		A	B	a1	a2	b1	b2
		index	(1)	(2)	(3)	(4)	(5)	(6)
ROOT	A	(1)	x					
ROOT	B	(2)		x				
ATOM	a1	(3)	x		x	x		
ATOM	a2	(4)	x		x	x		
ATOM	b1	(5)		x	x		x	
ATOM	b2	(6)		x		x	x	x

(b) An additional view of the DSM of the example schema, shown as the transpose of the initial diagram, and in Inputs in Columns (IC) form such that the source event is along the column and the target event is along the row. Additionally, all events exhibit an interaction along the diagonal, and the number of interactions is not indicated in the diagram. This is an interim step in preparation for the clustering algorithm.

Event type	Event name		B	b1	b2	A	a1	a2
		index	(2)	(5)	(6)	(1)	(3)	(4)
ROOT	B	(2)	x					
ATOM	b1	(5)	x	x			x	
ATOM	b2	(6)	x	x	x			x
ROOT	A	(1)				x		
ATOM	a1	(3)				x	x	x
ATOM	a2	(4)				x	x	x

(c) Clustering of the Design Structure Matrix (DSM) of the example schema reflecting potential groupings of events that exhibit close interactions. This algorithm was derived by Thebeau [162], and applied to the MP results using MATLAB[®].

Event type	Event name		B	b1	b2	A	a1	a2
		index	(2)	(5)	(6)	(1)	(3)	(4)
ROOT	B	(2)						
ATOM	b1	(5)	3				3	
ATOM	b2	(6)	3	3				3
ROOT	A	(1)						
ATOM	a1	(3)				3		1
ATOM	a2	(4)				3	3	

(d) Populating the Design Structure Matrix (DSM) with the number of interactions.

Figure 4.26. Example N-squared diagram and Design Structure Matrix (DSM) of the example schema, using a clustering algorithm. Adapted from [162].

4.2.2.1 Applying the DSM to MP results

Since the output of MP is a Directed Acyclic Graph (DAG), a directed graph without loops, the MP results may be used to directly populate the DSM. As demonstrated in Appendix C, the author wrote a MATLAB® script that builds the DSM, named “MPbuildN2.m” that conducts the following steps listed below and illustrated in Figure 4.27:

- Parses the MP output file written in .json format to a structured array in MATLAB®,
- Derives the unique labels of each event, termed as a node in this analysis,
- Tabulates the inclusion and precedence relationships of the behavior model, termed as an edge in this analysis, and
- Produces tabular results for further analysis.

Further analysis consists of 1) comparing multiple runs of MP to identify whether new relationships form at a higher scope of execution, and 2) executing a cluster algorithm to identify interactions that occur frequently and in proximity. This clustering analysis potentially identifies areas where the developer may wish to closely group design elements.

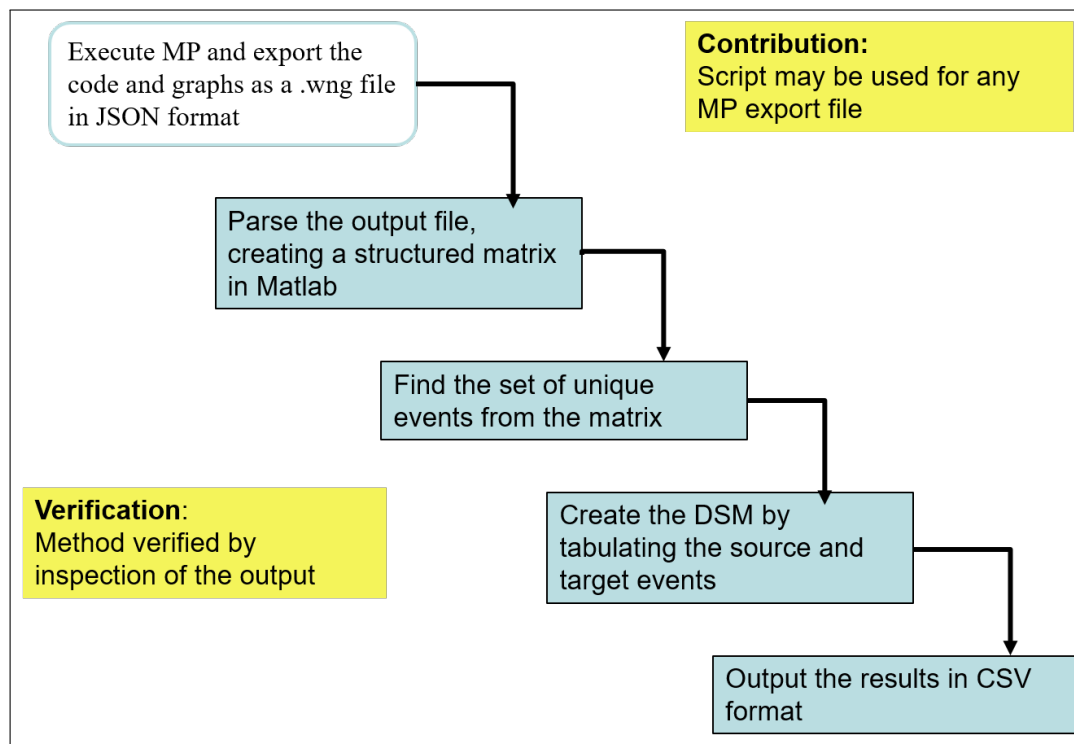


Figure 4.27. Derivation of the DSM using the MATLAB® script.

4.3 Verification of the Methodology

Verification of the methodology was accomplished by demonstration of two models 1) a decision model that captures a leader and subordinate interaction during a stressful procedure and 2) an object-to-effect model that captures system of systems functions of sensing, processing, and acting, based on the existence of some object of interest.

4.3.1 Derivation of the Decision Model

The Decision Model outlines a potential failure mode experienced in a hierarchical workforce, dealing with the perception of a routine procedure that leads to a critical decision by the leader. Inspired by a leadership book by Syed [17], this model applies to a surgical crisis and aviation mishap scenario that followed a similar fault pattern. The code and output traces are provided in Appendix A.1.

The Decision Model demonstrates the following:

- Uses Monterey Phoenix (MP) to model decision patterns based on a behavior narrative, with consistency across medical and aviation examples.
- Illustrates that an abstract architecture captures behaviors of human response within stressing environmental conditions.
- Verifies the methodology, outlined in Section 4.1, using a simple MP example, indicating criteria to repeat a step, to move forward, or to step backward.
- Employs constraints to shape the system outcomes. Types of constraints include logical, definition, simplification, and design.
- Incorporates interactions and constraints to formally define system behavior requirements.
- Employs an assertion checking function to identify pattern matching and then derives templates for particular behaviors of the system. Subsequent outcomes, executed at higher a scope of execution consist of combinations of these templates.
- Derives universal templates to identify pattern segments, including the successful and failed sequences, warnings, and the OODA loop.

- Verifies the method of pattern identification, based on the architecture structure (i.e., topology), combined with event semantics.
- Verifies the effects of increased scope and the identification of these previously defined templates across all possible outcomes.
- Uses the behavior model to derive a Bayesian belief network.
- Derives the probability of occurrence of each trace of the model.
- Implements an automated algorithm to trim branches of the decision logic, based on the constraints.
- Implements an automated algorithm to derive a Design Structure Matrix (DSM) that enables the developer to investigate both frequent and rare occurrences of patterns of behavior.

Derivation of the Decision Model follows the approach as outlined in the Section 4.1. Simulating the use of a repository, the model was written at an abstraction-level with direct application to both a surgical environment and aircraft landing environment, as illustrated in Figure 4.28. The type-specific templates were derived specifically from models that match the construct of a decision made by a leader with support by one or more subordinates. The universal templates have a general usage and may apply to other model types as well.

4.3.1.1 Decision Model: Behavior narrative

Syed [17] outlined two distinct cases, showing consistent decision-making patterns leading to failures, within a surgical procedure and an aviation mishap. The narratives are summarized as follows:

4.3.1.1.1 Medical procedure narrative: On March 29, 2005, a woman undergoes a routine surgical procedure. While under anesthesia, a rare side-effect occurs, causing a restriction in her airway. The surgeon immediately focuses on procedures to insert a breathing tube through her airway, but fails to succeed. The surgeon continues to focus on the airway tube, losing time. A nurse correctly observes the situation and retrieves a tracheotomy kit, that would enable an alternate airway and thereby alleviate the problem. The nurse recommends this solution to the surgeon, but

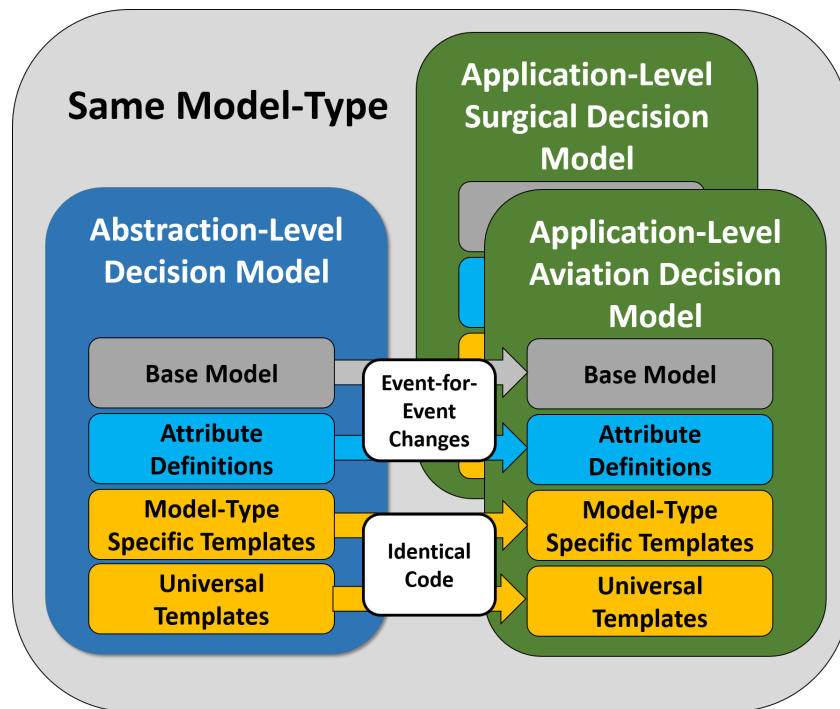


Figure 4.28. Universal and model-type specific templates used identically across the abstraction-level and application-levels.

this advice is not heeded by the surgeon. Time lapses, the patient loses oxygen, lapses into a coma and eventually dies [17], [178].

4.3.1.1.2 Aviation mishap narrative: On December 28, 1978, United Airlines Flight 178 attempts a routine landing in Portland, Oregon. Upon descent, the pilot notices that the landing gear light does not give a positive indication that the gear is locked in position. The pilot immediately enters a holding pattern so that he can investigate the situation and attempts to understand the landing gear status: whether locked or unlocked. As time lapses, the flight engineer notices that the fuel level is reaching a critically low state. The flight engineer indicates the condition to the pilot several times. The pilot fails to heed the engineer's recommendation until it is not possible to reach the runway. Engines fail upon approach, causing a crash landing, killing ten people, including the flight engineer [17], [179].

4.3.1.1.3 Consistent patterns: As illustrated in Figure 4.29, consistency within the decision process becomes readily discernible, even by simply comparing the narratives. A hierarchical structure failed to support awareness of the truly critical scenario facing the participants. The leader, or decision maker, became overly focused on what he perceived to be the problem, but the

conditions changed over time, causing a failure to reach the desired end-state. For this dissertation, a pattern is defined as a particular set of choices that leads an outcome or final state.

From these narratives, the model developer expands each of the relevant events, to consider all possible alternatives. The developer then adds interactions and constraints, thereby identifying the relevant patterns of behavior. The following paragraphs apply each step of the methodology, as outlined in Section 4.1.

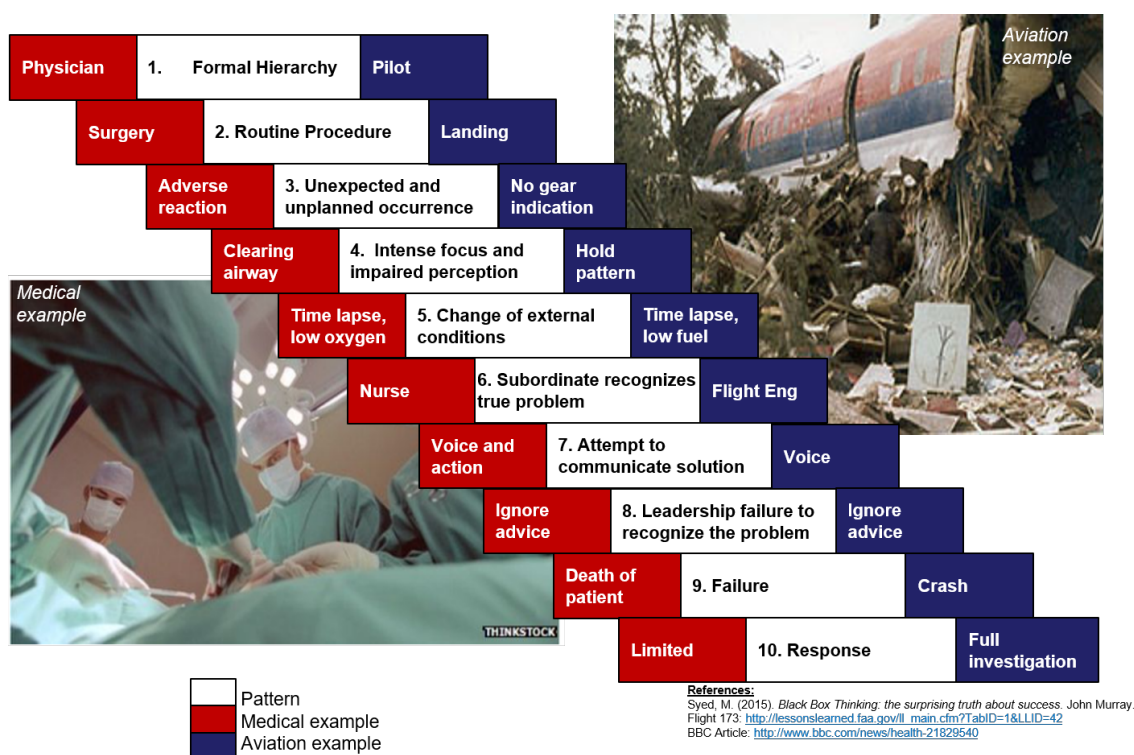


Figure 4.29. Decision pattern, showing a consistent pattern for medical and aviation examples. Adapted from [17]. Photograph sources: [178] and [179].

4.3.1.2 Decision Model: Event identification

Within this MP schema, root events include the leader (surgeon or pilot), the subordinate(s) (nurse or flight engineer), and the environment (problems associated with the routine surgery and landing). The routine procedure is shared among the leader, subordinate, and environment. The leader and subordinate have an independent perception that either correctly recognizes the environment or does not recognize the environment. The subordinate decides whether to communicate their individual perception to the leader, and the subordinate perception could be either correct or incorrect. The leader then has a context that receives input from a subordinate or rejects that input. The leader

then makes either a correct or incorrect decision. Finally, the environment includes the occurrence of a problem, or no problem; and provides an indication of a successful or failed outcome.

The structure of the events is listed as follows, with each root event listed at the top of the hierarchy. In each of the following sections, an abstracted, or generalized, list of events is described, followed by the specific application to the aviation or surgical crisis. The following lists the description of each set of events, using a hierarchy derived from the narrative based on each of the main objects in the narrative: the environment, the leader, and subordinate, as follows:

- **Environment**
 - **Problem:** The problem may exist or not exist
 - **Outcome:** The outcome may be a success or failure.
- **Subordinate**
 - **Perception:** The subordinate may correctly recognize the problem or not recognize the problem.
 - **Communication:** The subordinate decides whether to communicate the observation to the leader.
- **Leader**
 - **Perception:** The leader may correctly recognize the problem or not recognize the problem.
 - **Reception:** The leader may receive the communication by the subordinate, or fail to do so.
 - **Decision:** The leader decides on how to respond based on their own perception and the communication by the subordinate.
 - **Action:** The leader actions may be correct in accordance with the environment condition, or in opposition to the condition.

The following lists a derivation of the dependent events associated with each main object:

4.3.1.2.1 Environment events: As depicted in Figure 4.30 and the following MP code, the environment includes two events, each with two alternatives: 1) whether the problem exists, followed by 2) whether the desired result is attained. The figure illustrates the abstraction-level description, shown in blue, and the application of these events, written in terms of the aviation mishap and surgical crisis.

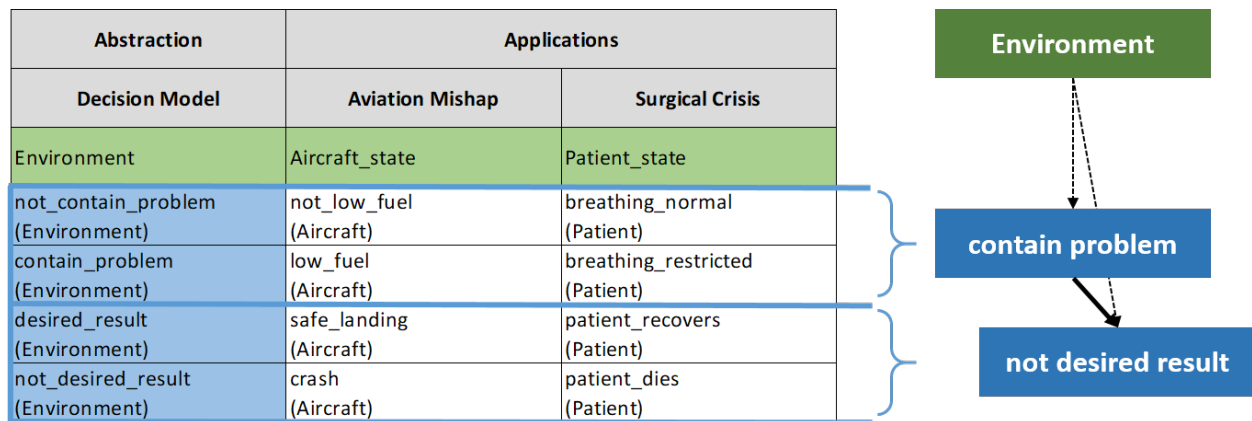


Figure 4.30. Decision Model events associated with the environment: the abstraction level includes whether the problem exists and whether the desired result is attained. Notice that the environment forms a ROOT event with inclusion relationships (dotted line) to the definition of the problem, and the outcome. A precedence relationship (solid line) exists between these events since the existence of a problem has an effect upon the result. An example of the possible combinations of events is shown on the right.

MP code segment 4.11: Events within the environment.

```

ROOT Environment: ( not_contain_problem
                    | contain_problem )
                    ( desired_result
                    | not_desired_result );

```

4.3.1.2.2 Leader events: Figure 4.31 outlines the four events associated with the leader, followed by the MP code listing to implement the set of events. Each of the following events have two alternatives: whether to recognize the environment, receive input, decide in accordance with the input received, and act in a way that is consistent with the environment.

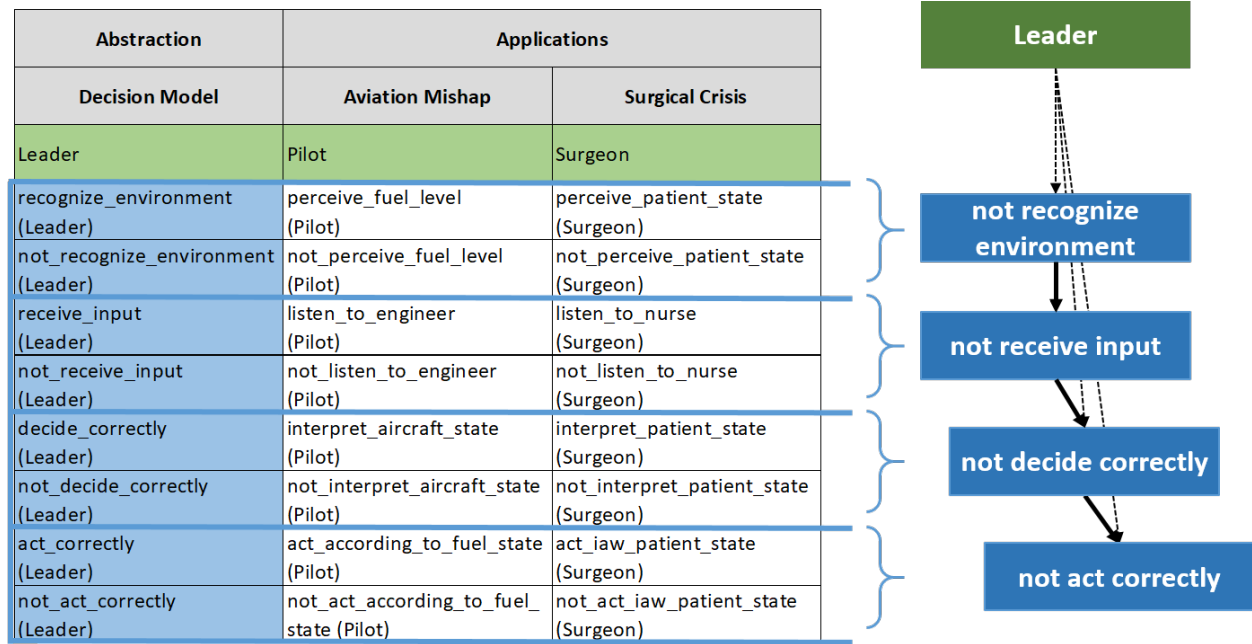


Figure 4.31. Decision Model events associated with the leader: The leader establishes a root event, with four dependent events. These dependent events are included in the set of actions possibly by the leader, each with two alternative as follows: whether the leader recognizes the environment, receives input, decides correctly, and acts correctly. The abstraction-level is shown in blue, with the application to the aviation mishap and surgical crisis outlined in white. An example of the possible combinations of events is shown on the right.

MP code segment 4.12: Events accomplished by the leader.

```

ROOT Leader:  ( recognize_environment
                | not_recognize_environment )

               ( receive_input
                | not_receive_input )

               ( decide_correctly
                | not_decide_correctly )

               ( act_correctly
                | not_act_correctly );

```


4.3.1.2.3 Subordinate events: As illustrated in Figures 4.32 and 4.33, each subordinate has a set of two dependent events: (1) whether to recognize the environment and (2) whether to communicate with the leader. Following these figures, the MP code adjusts the number of subordinates, as indicated by the number between the “<” and “>” symbols.

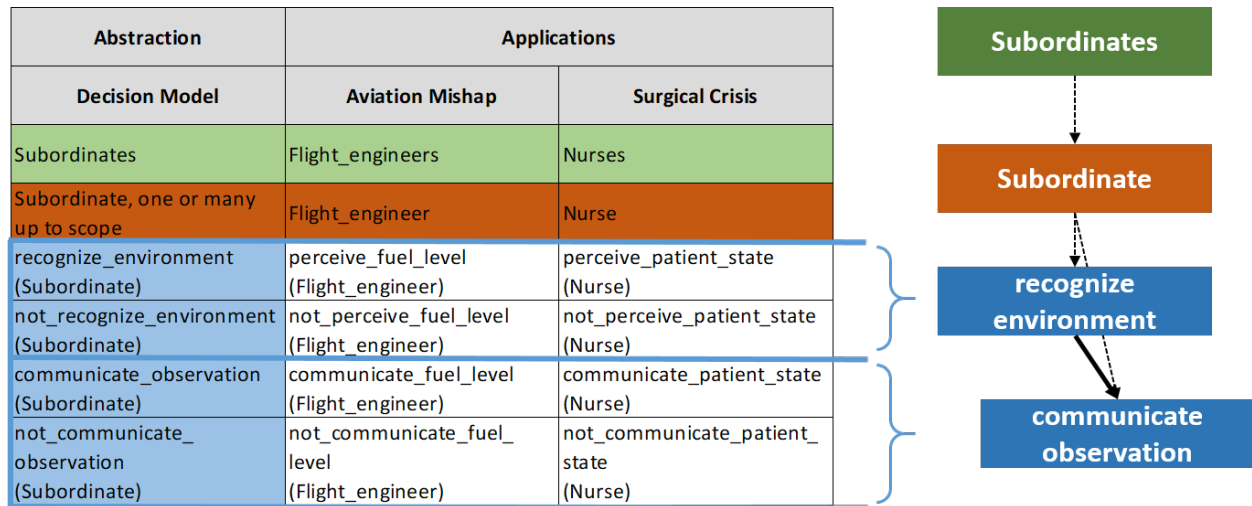


Figure 4.32. Decision Model events associated with the subordinate(s): Subordinates establishes a root event, such that one or many subordinates may be employed in the model. Each subordinate is a composite of a set of events, illustrated in orange. The dependent events include whether each subordinate recognizes the environment and communicates to the leader. The abstraction -level is shown in blue, with the applications outlined in white. An example of the possible combinations of events is shown on the right.

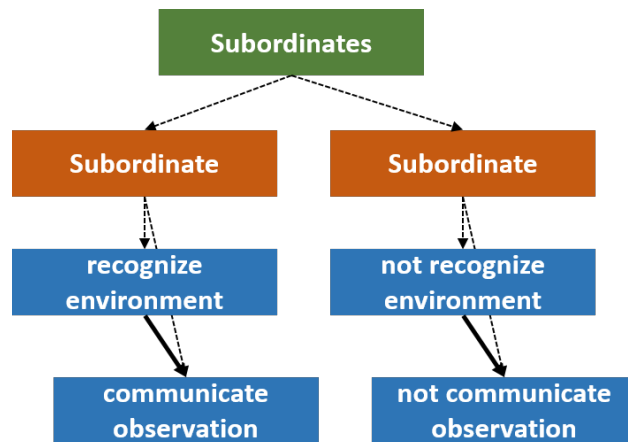


Figure 4.33. Decision Model with multiple subordinates.

MP code segment 4.13: Events accomplished by each subordinate.

```

ROOT Subordinates: { + <1> Subordinate + };
    Subordinate: ( recognize_environment
                  | not_recognize_environment )
                ( communicate_observation
                  | not_communicate_observation );

```

4.3.1.3 Decision Model: Coordination definition

Figure 4.34 illustrates the coordination across each set of root events. The coordination of the model identifies the interaction of the leader, each subordinate, and environment. These interactions include the following;

1. The problem state precedes the perception of both the leader and subordinate.
2. Communication by the subordinate precedes the leader's interpretation of that communication.
3. An action by the leader leads to an outcome.

The following sections outline the MP code associated with each interaction.

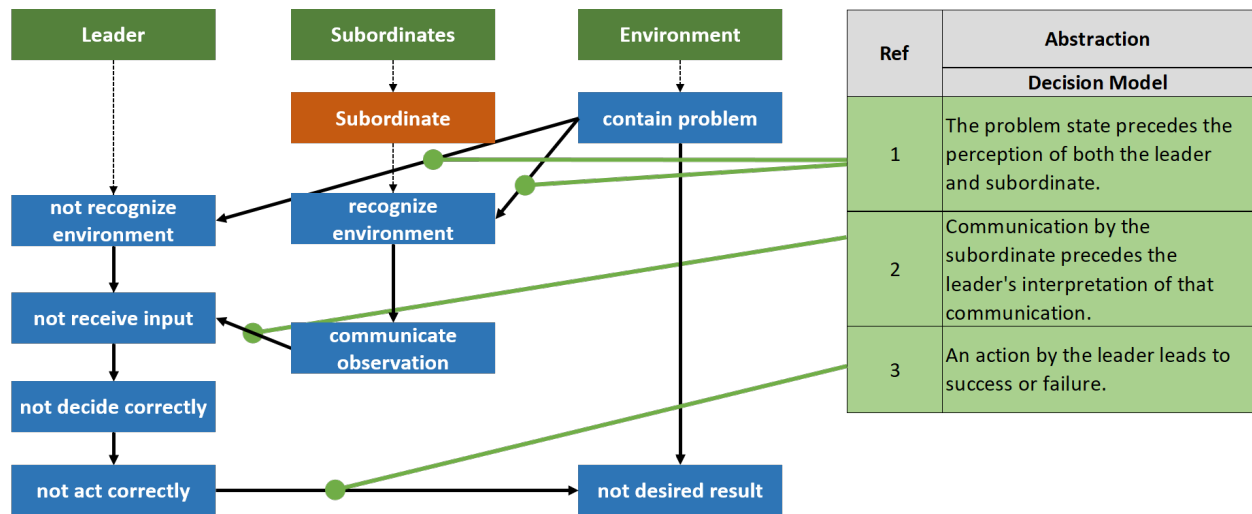


Figure 4.34. Decision Model coordination among the environment, leader, and subordinates defines the precedence relationships across the relevant events.

4.3.1.3.1 Coordination of the environment with the leader and subordinates: The existence of a problem state precedes the recognition of the problem. The leader and all of the subordinates share a common definition of the recognition of the problem, and so one coordination script addresses all associated occurrences of the perception, as follows:

MP code segment 4.14: Leader and subordinate perception of the problem state.

```

/** Interaction 1: The problem state precedes the perception */
/**
COORDINATE $x: ( not_contain_problem
                  | contain_problem )
DO COORDINATE
    $y: ( recognize_environment
          | not_recognize_environment )
DO ADD $x PRECEDES $y; OD; OD;

```

4.3.1.3.2 Coordination between the leader and subordinate: Communication by the subordinate to the leader is indicated as follows:

MP code segment 4.15: Communication between the leader and subordinate.

```
/**Interaction 2: Communication by the subordinate precedes */
/* the leadership interpretation of that communication */
/**/
COORDINATE $a: ( communicate_observation
                 | not_communicate_observation )

DO COORDINATE $b: ( receive_input
                  | not_receive_input )

DO ADD $a PRECEDES $b; OD; OD;
```

4.3.1.3.3 Coordination between the leader and the environment: The leader decides to act or not act correctly, and thereby producing an outcome, as follows:

MP code segment 4.16: Effect of leader on the environment.

```
/**Interaction 3: An action leads an outcome */
/**/
COORDINATE $a: ( act_correctly
                 | not_act_correctly),
               $b: ( desired_result
                  | not_desired_result )
DO ADD $a PRECEDES $b; OD;
```

4.3.1.3.4 A Functional Flow Block Diagram (FFBD) of the Decision Model: With the definition of event alternatives and interactions among the environment, leader, and subordinate(s), the FFBD can be constructed, as shown in Figure 4.35. This figure depicts each of the alternative events and hierarchy of the model structure.

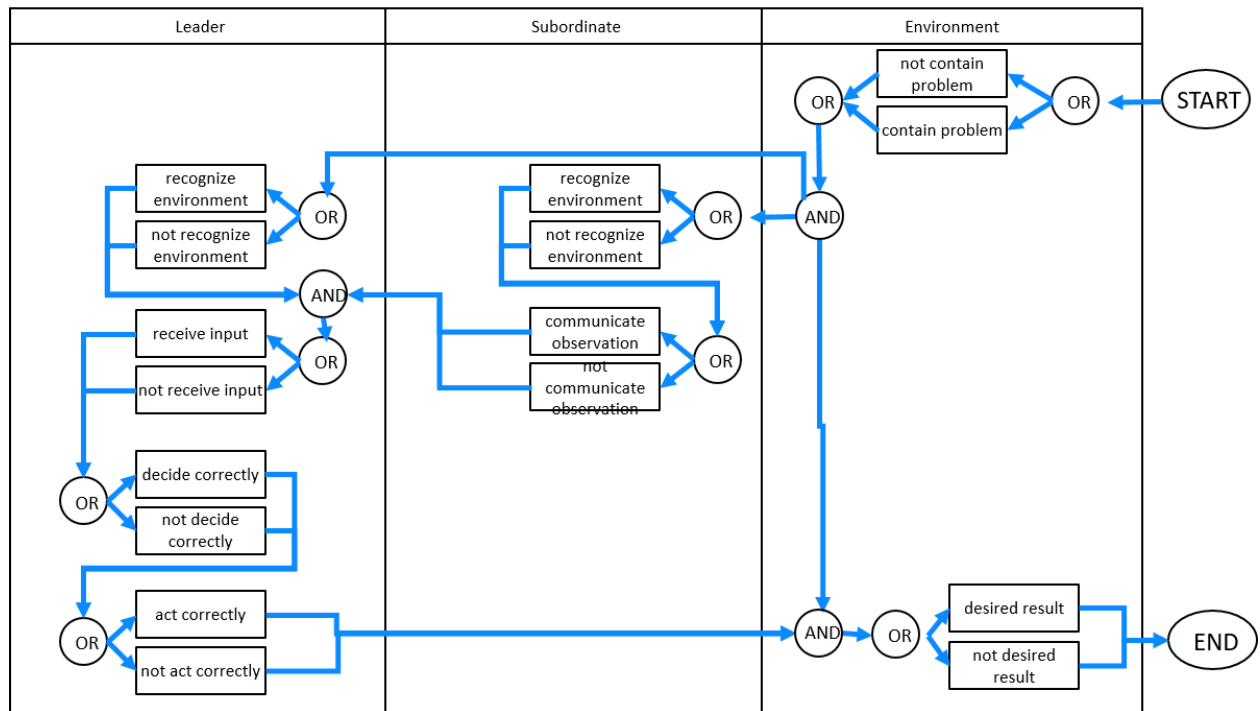


Figure 4.35. FFBD of the Decision Model.

4.3.1.4 Decision Model: Constraint derivation

As defined to this point, the model consists of eight events, each with two alternatives, and so the number of possible traces is (2^8) or 256. As illustrated in Figure 4.36 and Table 4.2, the constraints operate on the interactions among the events in the model. When applied, they reduce the number of traces from 256 to just twelve, as illustrated in Figure 4.37.

These constraints were derived interactively with model execution. This iterative process enables the developer to eliminate logical inconsistencies, enable simplification, derive design requirements, and apply definitions.

1. *Type: Logical* - If there is no problem in the environment, then always have a successful outcome.
2. *Type: Logical* - If all subordinates do not communicate, then the leader receives no input.
3. *Type: Simplification* - If the leader recognizes the environment and does not receive input, then the leader makes a correct decision.
4. *Type: Simplification* - Not receiving an input leads to an incorrect decision, and its corollary that receiving an input leads to a correct decision.
5. *Type: Simplification* - A correct decision leads to a correct action and its corollary that an

incorrect decision leads to an incorrect action.

6. *Type: Definition* - A correct action leads to a successful outcome, and its corollary that an incorrect decision leads to a failed outcome.

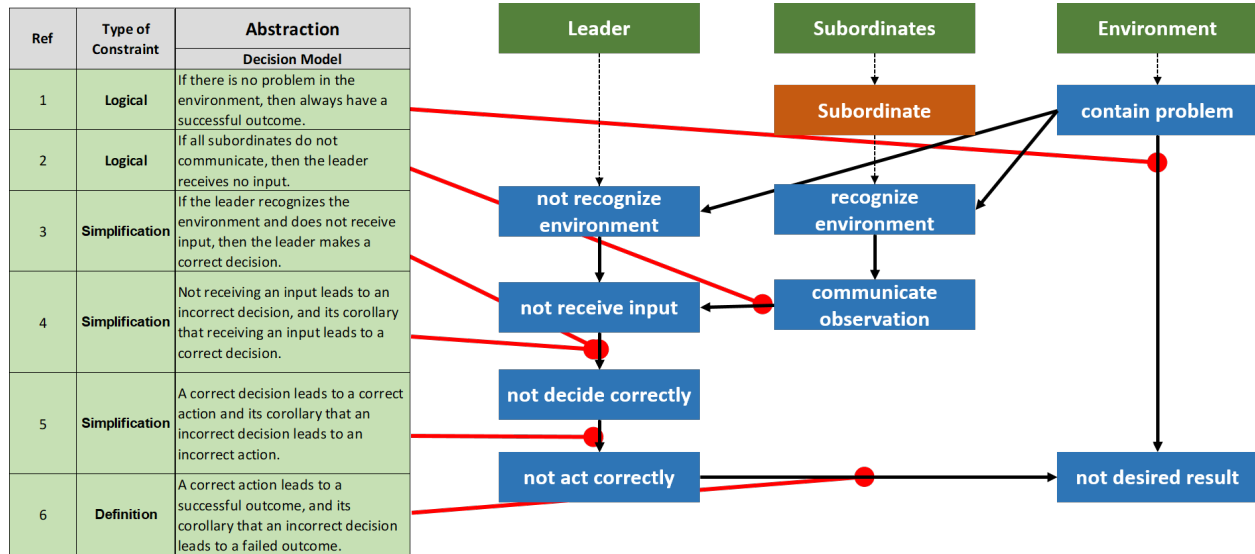


Figure 4.36. Decision Model constraints applied to the interactions of the model.

Table 4.2. Decision Model constraints as applied to the aviation mishap and surgical crisis applications.

Ref	Type of Constraint	Abstraction	Applications		Number of traces after cumulative constraints applied
		Decision Model	Aviation Mishap	Surgical Crisis	
0	none	n/a	n/a	n/a	256
1	Logical	If there is no problem in the environment, then always have a successful outcome.	If there is no problem in the aircraft then always have a successful outcome.	If there is no problem with the patient, then always have a successful outcome.	192
2	Logical	If all subordinates do not communicate, then the leader receives no input.	If all flight engineers do not communicate, then the pilot receives no input.	If all nurses do not communicate, then the surgeon receives no input.	144
3	Simplification	If the leader recognizes the environment and does not receive input, then the leader makes a correct decision.	If the pilot recognizes the landing gear and does not receive input, then the pilot makes a correct decision.	If the surgeon recognizes the patient airway and does not receive input, then the surgeon makes a correct decision.	108
4	Simplification	Not receiving an input leads to an incorrect decision, and its corollary that receiving an input leads to a correct decision.	Not receiving an input from the flight engineer leads to an incorrect decision, and its corollary that receiving an input leads to a correct decision.	Not receiving an input from the nurse leads to an incorrect decision, and its corollary that receiving an input leads to a correct decision.	48
5	Simplification	A correct decision leads to a correct action and its corollary that an incorrect decision leads to an incorrect action.	Correct interpretation of a low fuel state leads to enacting emergency landing procedures, and its corollary	Correct interpretation of the patient breathing state leads to enacting an emergency tracheotomy procedure, and its corollary	24
6	Definition	A correct action leads to a successful outcome, and its corollary that an incorrect decision leads to a failed outcome.	A correct action leads to a safe landing and its corollary that an incorrect decision leads to a fatal crash.	A correct action leads to a patient recovery and its corollary that an incorrect decision leads to a patient fatality.	12

MP code segment 4.17: Constraints applied to the Decision Model.

```

/**Constraint 1. LOGICAL: If there is no contains_problem in */
/* the environment, then always have a successful outcome. */
/**/
ENSURE (#not_contain_problem FROM Environment == 1 ->
        #desired_result FROM Environment == 1);

/**Constraint 2. LOGICAL: If all subordinates do not */
/* communicate, then the leader receives no input. */
/**/
ENSURE (#not_communicate_observation FROM Subordinates -
        #Subordinate == 0 -> #not_receive_input FROM Leader == 1);

/**Constraint 3. SIMPLIFICATION: If the leader recognizes the*/
/* environment and does not receive input, then the leader */
/* makes a correct decision. */
/**/
ENSURE (#recognize_environment FROM Leader -
        #not_receive_input == 0 -> #decide_correctly == 1);

/**Constraint 4. SIMPLIFICATION: Not receiving an input leads*/
/* to an incorrect decision, and its corollary. */
/**/
ENSURE (#receive_input FROM Leader == 1 ->
        #decide_correctly FROM Leader == 1);
ENSURE (#not_receive_input FROM Leader == 1 ->
        #not_decide_correctly FROM Leader == 1);

/**Constraint 5. SIMPLIFICATION: A correct decision leads to */
/* a correct action, and its corollary. */
/**/
ENSURE (#decide_correctly == #act_correctly);
ENSURE (#not_decide_correctly == #not_act_correctly);

/**Constraint 6. DEFINITION: A correct action leads to a */
/* successful outcome, and its corollary. */
/**/
ENSURE(#act_correctly == #desired_result);
ENSURE(#not_act_correctly == #not_desired_result );

```

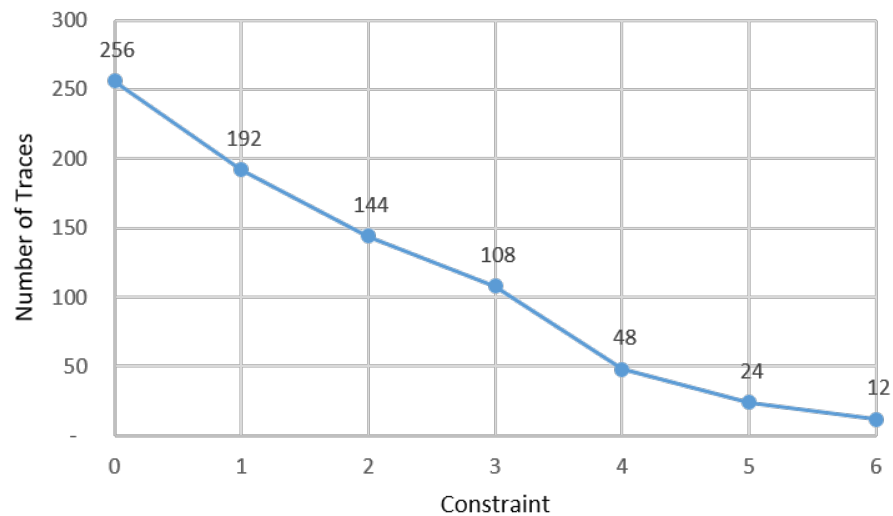



Figure 4.37. The effect of constraints on the Decision Model drastically reduces the number of valid traces, illustrated here with the scope of execution equal to one. For this model, the scope affects the number of subordinates represented in the model.

4.3.1.5 Decision Model: Pattern identification

This step of the methodology identifies the patterns that are inherent in the model, to include both model-specific patterns and universal patterns. Model-specific patterns apply to all models of this type, including the abstraction-level and application to the aviation mishap and surgical crisis. These patterns include each of the behaviors demonstrated at a scope of one, plus additional patterns observed at an increased scope.

Conversely, universal patterns include short-segments of behaviors to include sequences of positive and negative events, the OODA loop, transmit and receive events, and general statistics including identification of those events not in a pattern. Identification of these events that do not fit into a known pattern provides a means for the developer to identify new behaviors that may have been otherwise unnoticed.

Templates are employed to identify both the model-specific and universal patterns. These templates operate on attributes of each event, as defined in the following sections.

4.3.1.5.1 Semantic attributes: As listed in Table 4.3, attributes enable a simple notation that capture the inherent meaning of each of the events. An event may have one or more attribute associated with it, and each of the attributes listed in the table denote a Boolean expression of the

particular attribute. Note that the identical attribute is consistent at both the abstraction-level model and the applied models. In this way, templates that operate on these attributes are effective across the range of applicable problems.

Table 4.3. Decision Model attributes enable a simple notation that capture the inherent meaning of each of the events. Each of the attributes listed in the table denote a Boolean expression of the associated event.

Attribute	Description	Application of the Attribute to Events		
		Abstraction	Applied Models	
		Object-To-Effect	Target-to-Hit	Victim-to-Rescue
Ex, nEx	Existence or non-existence of the problem	contain_problem not_contain_problem (Environment)	low_fuel not_low_fuel (Aircraft_state)	breathing_restricted breathing_normal (Patient_state)
Se, nSe	Sense or not sense: Indication of whether the problem is perceived	perceive_environment not_perceive_environment (Leader, Subordinate)	perceive_fuel_level not_perceive_fuel_level (Pilot, Flight_Engineer)	perceive_patient_state not_perceive_patient_state (Surgeon, Nurse)
Ob, nOb	Observe or not observe: synonymous with sense, normally associated with a human observer	perceive_environment not_perceive_environment (Leader, Subordinate)	perceive_fuel_level not_perceive_fuel_level (Pilot, Flight_Engineer)	perceive_patient_state not_perceive_patient_state (Surgeon, Nurse)
Or, nOr	Orient or not orient: based on perception and communication	receive_input not_receive_input (Leader)	listen_to_engineer not_listen_to_engineer (Pilot)	listen_to_nurse not_listen_to_nurse (Surgeon)
De, nDe	Decide on not decide: based on processing available data	decide_correctly not_decide_correctly (Leader)	interpret_aircraft_state not_interpret_aircraft_state (Pilot)	interpret_patient_state not_interpret_patient_state (Surgeon)
Ac, nAc	Act or not act: based on the decision and the environment state	act_correctly not_act_correctly (Leader)	Conditional: land_aircraft not_land_aircraft (Pilot)	Conditional: conduct_tracheotomy not_conduct_tracheotomy (Surgeon)
Tx, nTx	Transmit or not transmit perception, communicate information	communicate_observation not_communicate_observation (Subordinate)	communicate_fuel_level not_communicate_fuel_level (Flight_Engineer)	communicate_patient_state not_communicate_patient_state (Nurse)
Rx, nRx	Receive or not receive the communication	receive_input not_receive_input (Leader)	listen_to_engineer not_listen_to_engineer (Pilot)	listen_to_nurse not_listen_to_nurse (Pilot)
favorable, unfavorable	Identification of positive or negative events	each event within the schema identified as favorable or unfavorable		

The following MP code defines each attribute and adds an indication of whether an event is within a template, along with a number associated with the number of events in each trace that are not in a template and the number of traces in the schema that include events not in a template.

MP code segment 4.18: Designation of the attributes associated with each event.

```

/* Attribute definitions indicate as Boolean expressions,
   that need to be derived from a repository, employing a defined
   ontology. */
ATTRIBUTES{bool is_authority, is_assistant, is_environ,
             favorable, unfavorable,
             Se, Ob, Or, De, Ac,
             nSe, nOb, nOr, nDe, nAc,
             Tx, Rx, Re, Int,
             nTx, nRx, nRe, nInt,
             EX, nEX,
             fail, success;
};

/* Attributes used to identify whether events in a template */
ATTRIBUTES {bool inTemplate; number count, tracecount;};

```

The following MP code declares the leader as the authority, the environment, and each of the root events and each subordinate to be within a template.

MP code segment 4.19: Attributes aligned to each of the root events.

```

/* Attribute assignments associated with each event */
Leader.is_authority := true;
Leader.inTemplate   := true;
Environment.is_environ := true;
Environment.inTemplate := true;
Subordinates.inTemplate := true;
COORDINATE $x: Subordinate DO $x.is_assistant := true; OD;

```

Attributes aligned to the Environment: As illustrated in Figure 4.38 and the following MP code, each event incorporates a relevant attribute derived from a repository. Note that the assignment of the attributes via the BUILD command shifts that event to a composite-type of event, changing its indication from blue to orange.

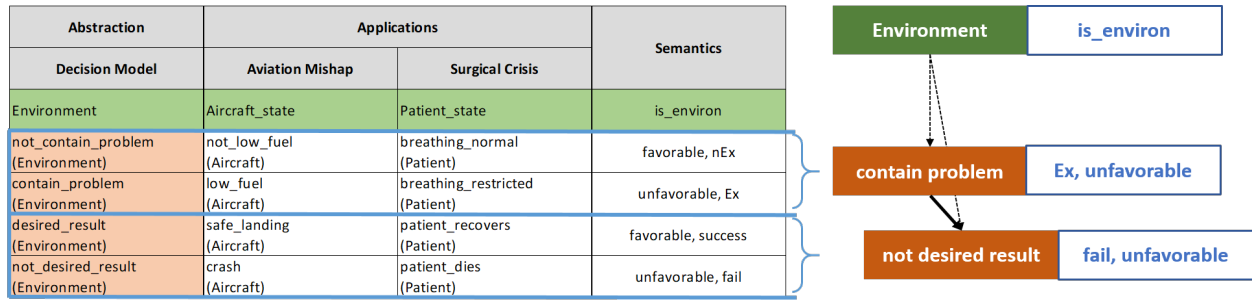


Figure 4.38. Decision Model attributes aligned to the set of environment events.

MP code segment 4.20: Environment attributes.

```

not_contain_problem :
    BUILD{ nEx := true;    favorable := true; };
contain_problem :
    BUILD{ Ex := true;    unfavorable := true; };

desired_result :
    BUILD{ success := true;    favorable := true; };
not_desired_result :
    BUILD{ fail := true;    unfavorable := true; };
  
```

Attributes aligned to the Leader: As illustrated in Figure 4.39, assign the attributes associated with the OODA loop and the receive segment of communication.

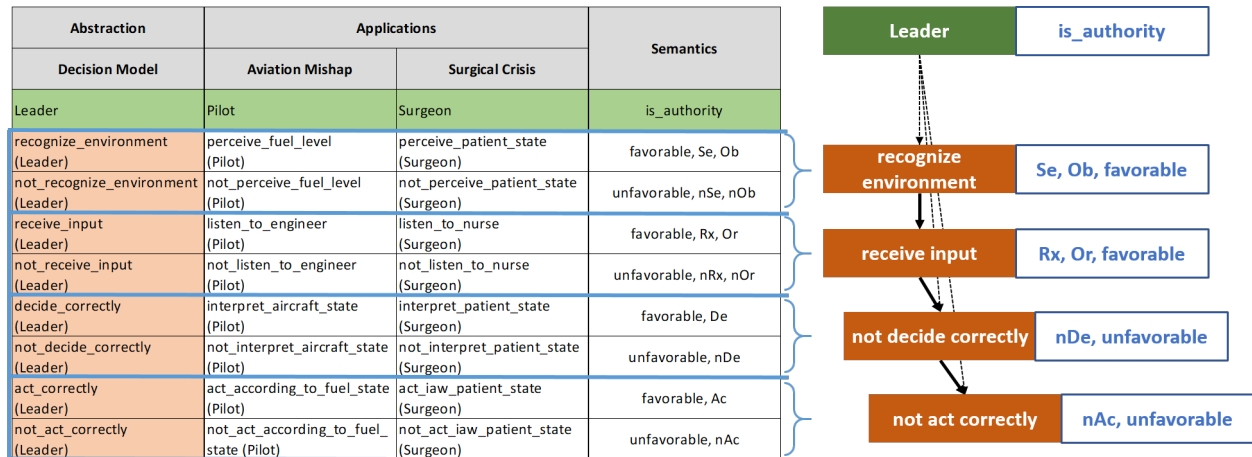


Figure 4.39. Decision Model attributes aligned to the set of leader events.

MP code segment 4.21: Leader attributes.

```

recognize_environment :
    BUILD{ Ob:= true; Se:= true; favorable := true; };
not_recognize_environment :
    BUILD{ nOb:= true; nSe:= true; unfavorable := true; };

receive_input :
    BUILD{ Rx:= true; Or:= true; favorable:= true; };
not_receive_input :
    BUILD{ nRx:= true; nOr:= true; unfavorable:= true; };

decide_correctly :
    BUILD{ De:= true; favorable:= true; };
not_decide_correctly :
    BUILD{ nDe:= true; unfavorable:= true; };

act_correctly :
    BUILD{ Ac:= true; favorable:= true; };
not_act_correctly :
    BUILD{ nAc:= true; unfavorable:= true; };

```

Attributes aligned to each subordinate: As illustrated in Figure 4.40, the model provides the attribute assignments to each subordinate. These attributes include the sensing or observation event and the transmit portion of communication.

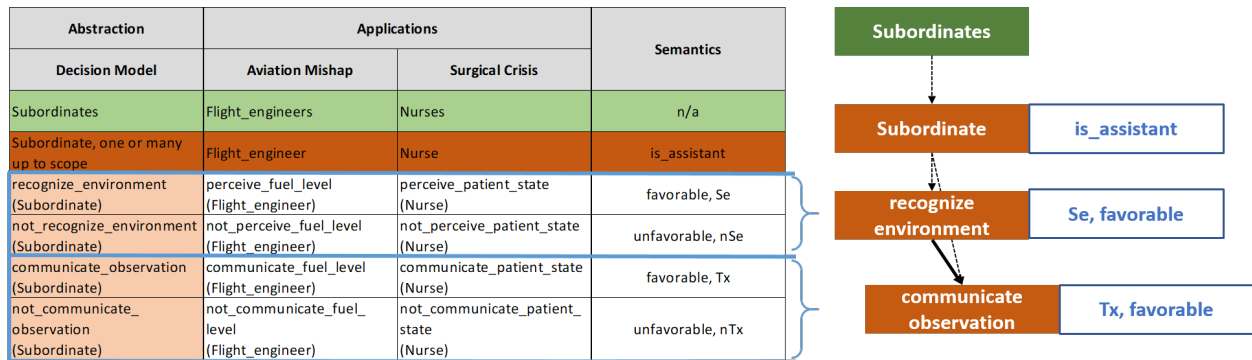


Figure 4.40. Decision Model attributes aligned to the set of events of each subordinate.

MP code segment 4.22: Subordinate attributes.

```

communicate_observation :
    BUILD{ Tx:= true; favorable:= true; };
not_communicate_observation :
    BUILD{ nTx:= true; unfavorable:= true; };

```

4.3.1.5.2 Topology: The topology of the Decision Model consists of the “lines and boxes” of the model. Figure 4.41 illustrates this topology with the associated attributes. This alignment enables

the developer to identify templates that capture the inherent patterns of the model.

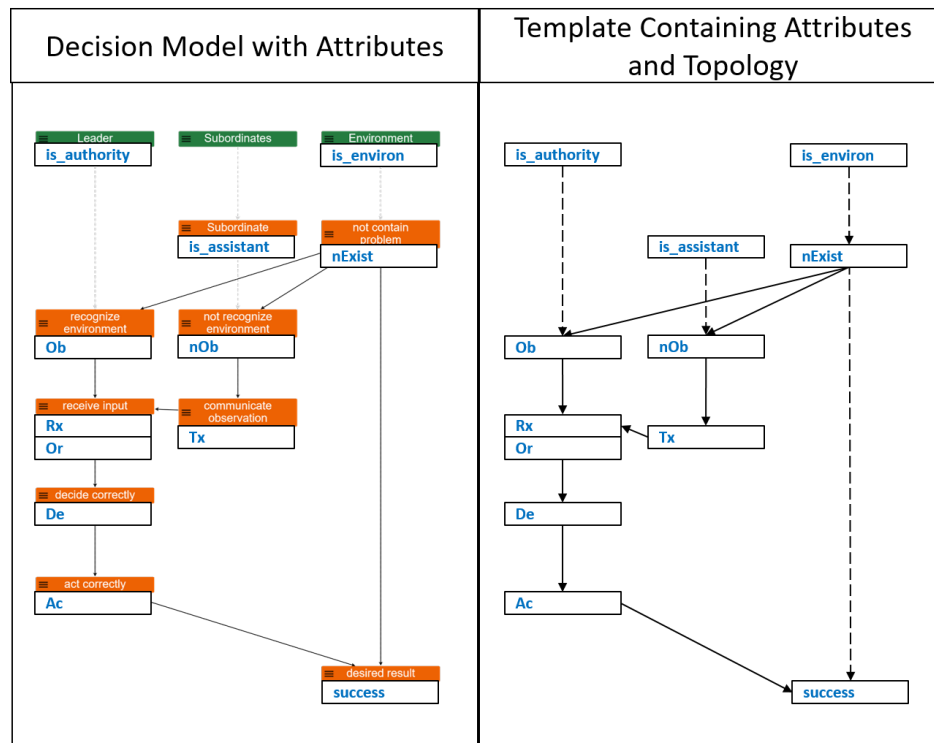


Figure 4.41. Decision Model topology as populated with the event attributes. This topology and alignment of the attributes, shown on the right, form the basis for template searches that identify inherent patterns of the model.

4.3.1.5.3 Templates: The model-specific templates apply to the abstraction level models and the application-level models of the aviation mishap and surgical crisis. The universal templates identify short sequences of behavior.

Model-specific templates: The following lists the patterns that are matched to each trace of execution, given a scope of one, such that there is one subordinate represented in the model. Trace number nine is representative of the series of events that led to the failures described in the narrative. These patterns are also illustrated in Appendix A.1.

1. Both leader and subordinate correctly perceive no problem.
2. Both leader and subordinate see a real problem.
3. Subordinate perceives a problem that does not exist.

4. Subordinate fails to see a real problem.
5. Leader perceives a problem that does not exist.
6. Leader incorrect perception, trusts subordinate.
7. Incorrect perception, but no problem exists.
8. Incorrect perception, but correct decision.
9. Leader fails to consider subordinate.
10. Subordinate fails to communicate problem.
11. Leader and Subordinate(s) are wrong with communication.
12. All went wrong.

MP code segment 4.23: Decision Model Template 9: Leader denies subordinate input.

```

/*****
/* Decision Model Template 9: Leader fails to consider
subordinate. */
**/
COORDINATE $authority:      $$EVENT SUCH THAT
                              $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                              $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.nRx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.nDe
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.nAc
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.fail
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 9:"); MARK;
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate        := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

Universal templates: These templates consist of short segments of behavior as identified by the event attributes. The following sections provide the MP code and resulting output trace for each of the following segments:

- Positive and negative events
- Transmit and receive
- OODA loop

Positive and negative events: The model identifies both favorable and unfavorable attributes, assigning each event to the indication of whether positive or negative and displaying this indication on each event in the trace, as illustrated in Figure 4.42 and facilitated by the following code:

MP code segment 4.24: Universal Template: Positive and negative events.

```
/*Positive events*/
COORDINATE $parent: $$EVENT
DO COORDINATE $positiveEvents: $$EVENT
    SUCH THAT $positiveEvents.favorable
DO
    IF ($positiveEvents IN $parent) THEN
        SAY ("Positive event: " $parent " does " $positiveEvents);
    FI; OD; OD;

/*Negative events*/
COORDINATE $parent: $$EVENT
DO COORDINATE $negativeEvents: $$EVENT
    SUCH THAT $negativeEvents.unfavorable
DO
    IF ($negativeEvents IN $parent) THEN
        SAY ("Negative event: " $parent " does " $negativeEvents);
    FI; OD; OD;
```

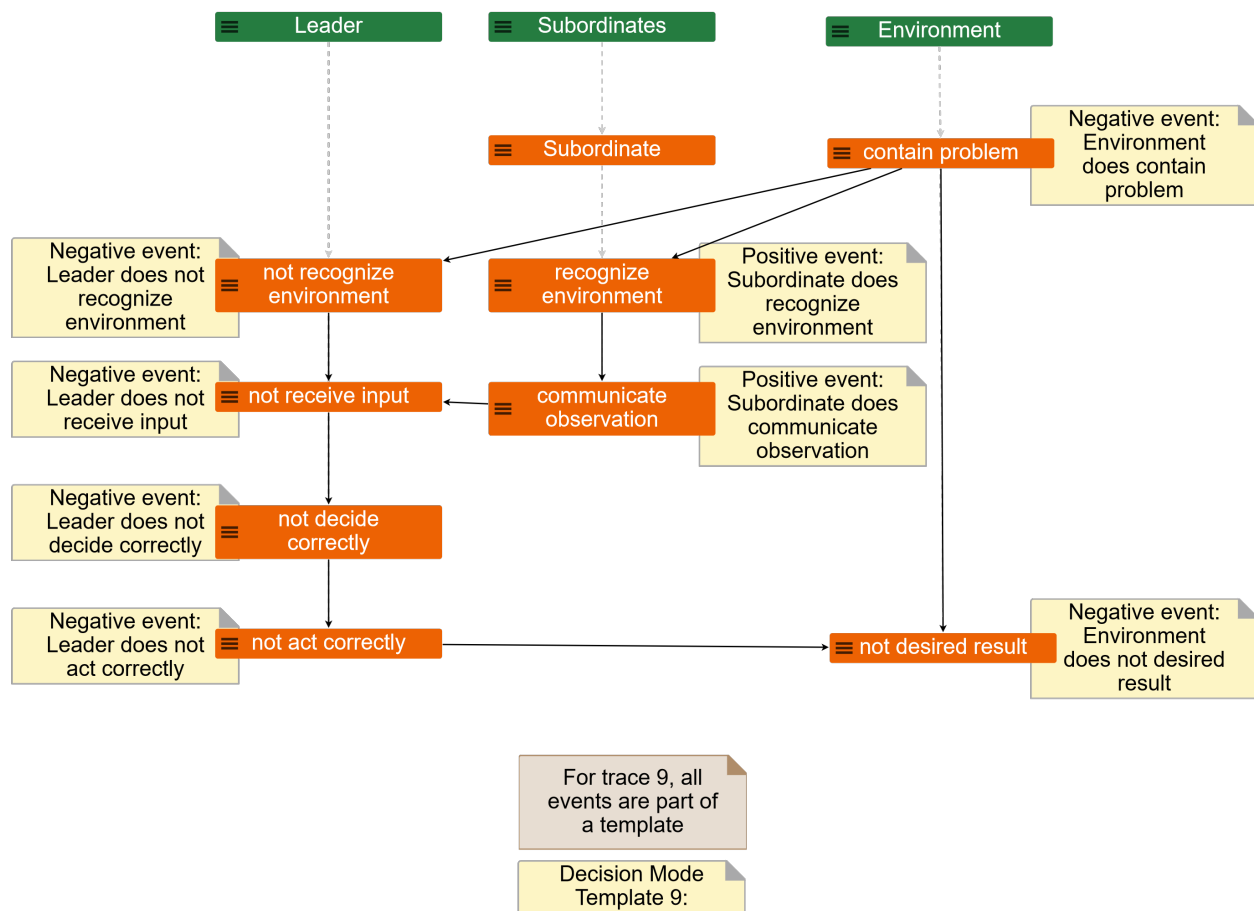



Figure 4.42. Decision Model identifying positive and negative events using universal templates, illustrating the Decision Model Template 9 using the associated the model-specific template. This template identifies the pattern that is consistent with the narrative of both the aviation mishap and surgical crisis.

Transmit and receive segment: This sequence occurs when the subordinate effectively communicates the perception which is received by the leader. This sequence occurs in Trace 8, as illustrated in Figure 4.43. These events were identified by the following template:

MP code segment 4.25: Universal Template: Transmit and receive.

```

/**
COORDINATE $transmit:  $$EVENT SUCH THAT $transmit.Tx
DO COORDINATE $receive:  $$EVENT SUCH THAT ($receive.Rx
                        AND $transmit PRECEDES $receive)
DO
  SAY("Transmit-Receive: Tx:" $transmit " , Rx:" $receive );
OD; OD;

```

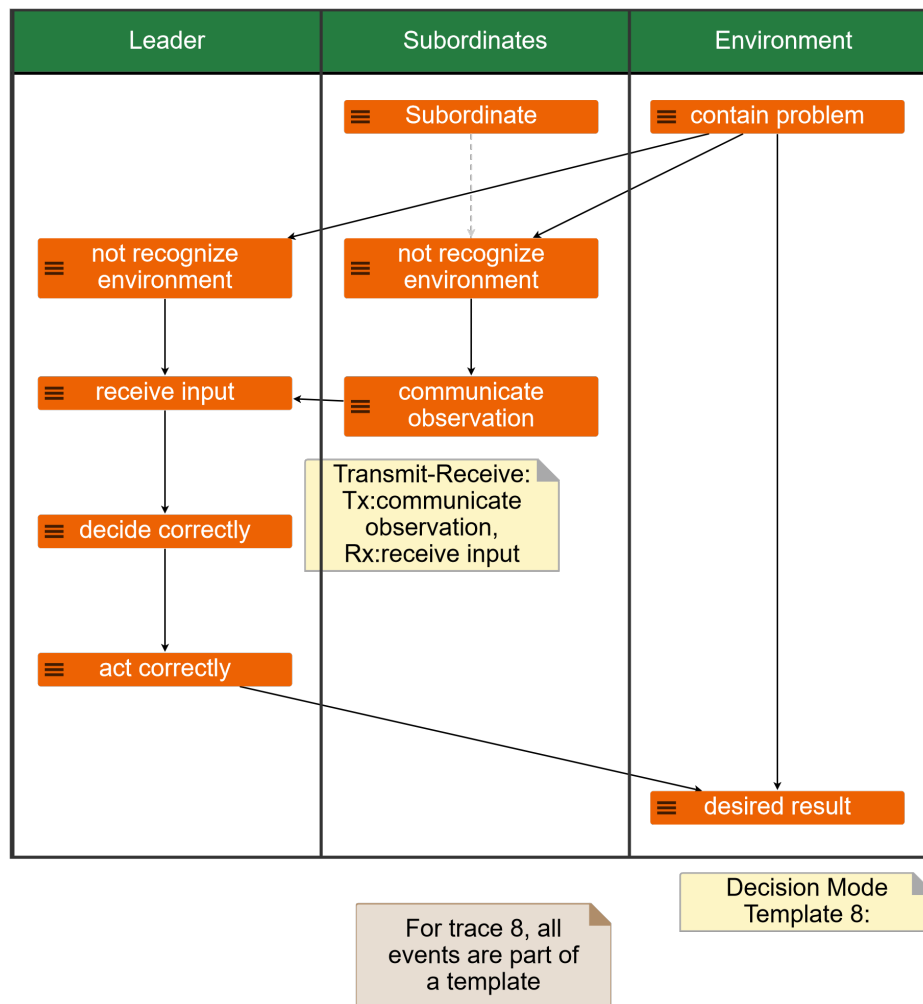


Figure 4.43. Decision Model, capturing the transmit and receive segments associated with the model-specific Template 8. In this pattern, the subordinate communicates the observation which is received by the leader.

The OODA loop: This sequence was identified when the problem is recognized, the leader orients to the perception, makes the correct decision, and acts according to the problem, as illustrated in Figure 4.44. Additionally, the model identified a coordinated OODA sequence in which there exists a send-receive series within the OODA. The following code outlines these templates:

MP code segment 4.26: Universal Template: Observe, Orient, Decide, Act (OODA) loop.

```

/* Identify the OODA LOOP */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $orient:  $$EVENT SUCH THAT ($orient.Or
                                AND $orient AFTER $observe)
DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.De
                                AND $decide AFTER $orient)
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
                                AND $act AFTER $decide)
DO
    SAY("OODA loop: " $observe ", " $orient ", " $decide ", "
        $act);
OD; OD; OD; OD;

```

MP code segment 4.27: Universal Template: Coordinated OODA loop.

```

/* Identify a Coordinated OODA */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
                                AND $transmit AFTER $observe)
DO COORDINATE $receive:  $$EVENT SUCH THAT ($receive.Rx
                                AND $receive AFTER $transmit)
DO COORDINATE $orient:   $$EVENT SUCH THAT ($orient.Or
                                AND $orient AFTER $observe)
DO COORDINATE $decide:   $$EVENT SUCH THAT ($decide.De
                                AND $decide AFTER $orient)
DO COORDINATE $act:      $$EVENT SUCH THAT ($act.Ac
                                AND $act AFTER $decide)
DO MARK;
SAY("Coordinated OODA loop: " $observe ", " $transmit ", "
    $receive ", " $orient ", "
    $decide ", " $act);
OD; OD; OD; OD; OD; OD;

```

Events not in a template: These events were identified by the following code, which is used to derive additional templates at higher scope, as outlined in the pattern analysis of section 4.3.1.6.

MP code segment 4.28: Universal Template: Identification of events not in a template.

```

/*Mark the traces that have events not in a template*****/
/**/
COORDINATE $a:$$EVENT SUCH THAT NOT $a.inTemplate
DO ADD SAY("not_in_template") not_in_template $a; MARK;
    GLOBAL.count += 1; count += 1;
OD;

```

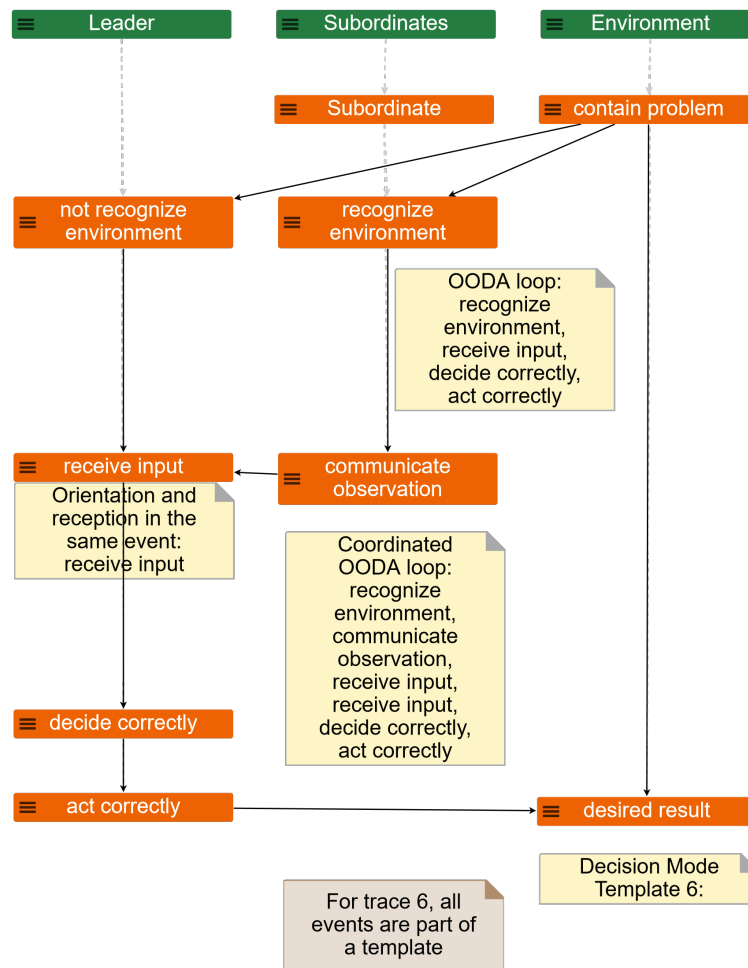


Figure 4.44. Decision Model, capturing the OODA segment such that the subordinate correctly identifies the problem, communicates with the leader, then the leader receives the input, decides and acts correctly. The model-specific pattern was identified by Decision Model Template 6.

4.3.1.6 Decision Model: Analysis

Analysis techniques applied to this model include the following:

- Identification of additional model-specific templates that occur at higher scope of execution.
- Determination of the probability of each trace.
- Determination of the Design Structure Matrix (DSM), enabling a comparison of results at higher scope of execution and analysis of the cluster algorithm.

4.3.1.6.1 The run matrix: As depicted in Table 4.4, the five runs varied the number of subordinates from one to five, consecutively. The total number of potential traces was 87,296 with 6,572 valid traces after applying the constraints to the model. All fourteen templates, listed in Table 4.5,

were derived within the first two runs, with all results from runs three through five consisting of combinations of these fourteen templates.

Table 4.4. Decision Model run matrix, indicating 6,572 valid traces after applying the constraints to the model.

Run #	Number of Subordinates	Number of Combinations of Alternatives for Each Set of Functions			Number of Potential Traces	Number of Valid Traces	
		Leader	Subordinates	Environment			
1	1	$2^4 =$	$2^2 =$	$2^2 =$	256	12	Templates 1-12
		16	4	4			
2	2	16	16	4	1,024	64	Combinations of Templates 1-12 and New Templates 13 or 14
3	3	16	64	4	4,096	288	Combinations of Templates 1-14
4	4	16	256	4	16,384	1216	
5	5	16	1024	4	65,536	4992	
Totals					87,296	6,572	

4.3.1.6.2 Pattern analysis: This analysis was implemented by increasing the scope of execution in order to identify whether additional templates are needed to capture patterns of behavior. For the Decision Model, increased scope of execution was enacted by increasing the number of subordinates. Upon execution of the model to include two subordinates, several events were found to not be captured by a template, as indicated in Figure 4.45. Note that in this case, one of the subordinates does not communicate, but that event precedes the leader's reception of communication. The developer needs to implement a new template with one of the following options:

1. Consider that there is a point in time in which the leader receives communication from one subordinate but not from another. Both events from each subordinated precede the leader's reception. Figure 4.46 illustrates that pattern, executed by the associated code, listed after the figure.
2. Limit the precedence relationship of the subordinate to the leader such that only the communication of one of the subordinates precedes the reception by the leader. In this case, the coordination logic between the leader and subordinate employs a conditional statement that prevents the subordinate event of "not communicate observation" from preceding the leader's event of "receive input." Figure 4.47 illustrates that pattern, executed by the associated code, listed after that figure.

The complete set of templates are listed in Table 4.5.

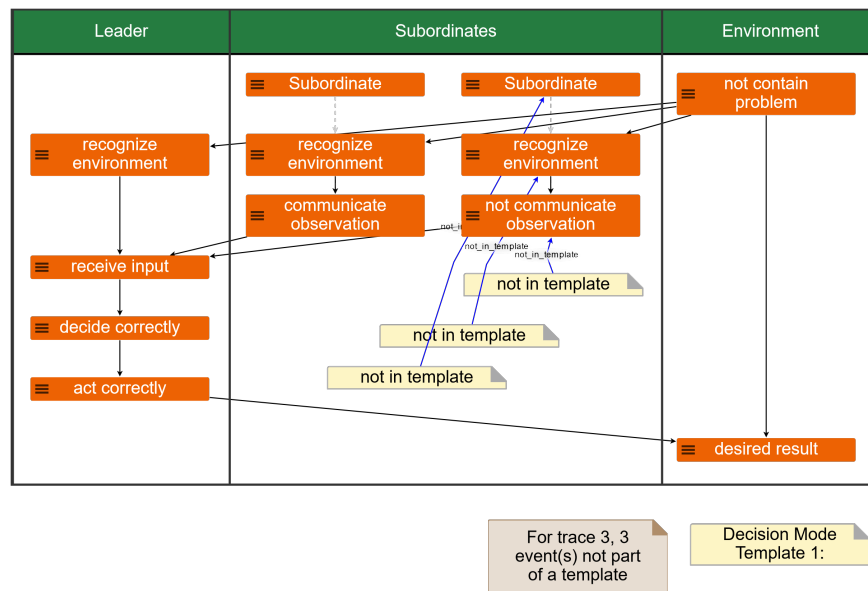


Figure 4.45. Additional subordinate with no supporting template.

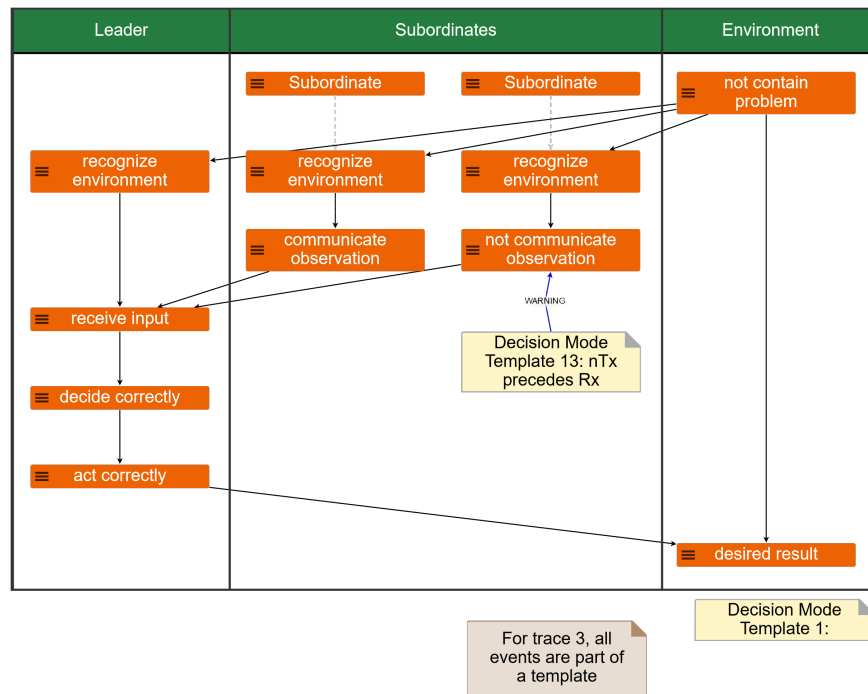


Figure 4.46. Additional subordinate with Template 13.

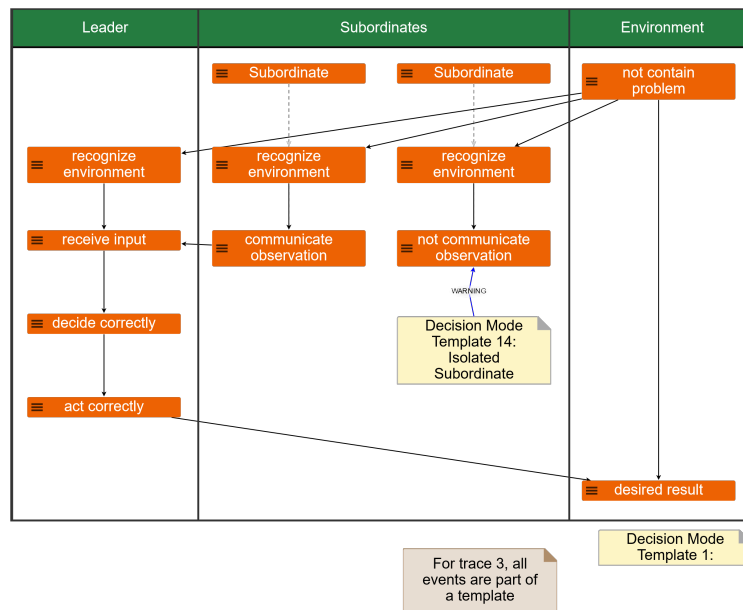


Figure 4.47. Additional subordinate with Template 14 implemented as a means to identify that the second subordinate has no interaction with the leader.

MP code segment 4.29: Decision Model Template 13.

```

COORDINATE $assistant1:  $$EVENT SUCH THAT
                           $assistant1.is_assistant
DO COORDINATE $assistant2:  $$EVENT SUCH THAT
                           ( $assistant2.is_assistant
                           AND NOT
                           $assistant2.inTemplate )
DO COORDINATE $perceive1:  $$EVENT SUCH THAT (
                           ( $perceive1.Se OR $perceive1.nSe)
                           AND $perceive1 IN $assistant1 )
DO COORDINATE $perceive2:  $$EVENT SUCH THAT (
                           ( $perceive2.Se OR $perceive2.nSe)
                           AND $perceive2 IN $assistant2 )
DO COORDINATE $transmit1:  $$EVENT SUCH THAT ($transmit1.Tx
                           AND $perceive1 PRECEDES $transmit1 )
DO COORDINATE $transmit2:  $$EVENT SUCH THAT ($transmit2.nTx
                           AND $perceive2 PRECEDES $transmit2 )
DO COORDINATE $receive:    $$EVENT SUCH THAT ($receive.Rx
                           AND $transmit1 PRECEDES $receive
                           AND $transmit2 PRECEDES $receive )
DO ADD SAY("Decision Model Template 13: nTx precedes Rx")
                           WARNING $transmit2; MARK;
  $assistant2.inTemplate   := true;
  $perceive2.inTemplate    := true;
  $transmit2.inTemplate    := true;
OD; OD; OD; OD; OD; OD; OD;

```

MP code segment 4.30: Decision Model Template 14.

```

COORDINATE $assistant1:  $$EVENT SUCH THAT
                           $assistant1.is_assistant
DO COORDINATE $assistant2:  $$EVENT SUCH THAT
                           ( $assistant2.is_assistant
                           AND NOT $assistant2.inTemplate )
DO COORDINATE $perceive1:  $$EVENT SUCH THAT (
                           ( $perceive1.Se OR $perceive1.nSe)
                           AND $perceive1 IN $assistant1 )
DO COORDINATE $perceive2:  $$EVENT SUCH THAT (
                           ( $perceive2.Se OR $perceive2.nSe)
                           AND $perceive2 IN $assistant2 )
DO COORDINATE $transmit1:  $$EVENT SUCH THAT ($transmit1.Tx
                           AND $perceive1 PRECEDES $transmit1 )
DO COORDINATE $transmit2:  $$EVENT SUCH THAT ($transmit2.nTx
                           AND $perceive2 PRECEDES $transmit2 )
DO COORDINATE $receive:    $$EVENT SUCH THAT ($receive.Rx
                           AND $transmit1 PRECEDES $receive
                           AND NOT $transmit2 PRECEDES $receive )
DO
  ADD SAY("Decision Mode Template 14: Isolated " $assistant2)
                           WARNING $transmit2; MARK;
  $assistant2.inTemplate   := true;
  $perceive2.inTemplate    := true;
  $transmit2.inTemplate    := true;
OD; OD; OD; OD; OD; OD; OD;

```


Table 4.5. Decision Model templates 1 through 12 were identified at a scope of one, with only one subordinate. Templates 13 and 14 were derived at a scope of two, indicated as warnings due to unique scenarios when the leader receives communication from at least one subordinate, but does not receive communication by an alternate subordinate. These fourteen templates capture behaviors at execution at a scope greater than two, with more than two subordinates.

Type	Ref.	Abstraction	Applications	
		Decision Model	Aviation Mishap	Surgical Crisis
Model-specific Templates	1	Both leader and subordinate correctly perceive that no problem exists	Both pilot and flight engineer correctly perceive a normal fuel state	Both surgeon and nurse correctly perceive normal patient breathing
	2	Both leader and subordinate correctly perceive that a problem exists	Both pilot and flight engineer correctly perceive a low fuel state	Both surgeon and nurse correctly perceive restricted patient breathing
	3	Subordinate incorrectly perceives a problem	Flight engineer perceives a low fuel state that does not exist	Nurse perceives restricted patient breathing that does not exist
	4	Subordinate fails to recognize an existing problem	Flight engineer fails to recognize an existing low fuel state.	Nurse fails to recognize the existence of restricted patient breathing
	5	Leader perceives a problem that does not exist	Pilot perceives a low fuel state that does not exist	Surgeon perceives restricted patient breathing that does not exist
	6	Leader fails to recognize an existing problem, but trusts subordinate perception	Pilot fails to recognize a low fuel state, but trusts flight engineer perception	Surgeon fails to recognize the existence of restricted breathing, but trusts nurse perception
	7	Incorrect perception, but no problem exists	Pilot and flight engineer incorrectly perceive low fuel level	Surgeon and Nurse incorrectly perceive restricted patient breathing
	8	Incorrect perception, but correct decision.	Pilot and flight engineer incorrectly perceive fuel level, but correct decision	Surgeon and Nurse incorrectly perceive patient breathing, but correct decision
	9	Leader fails to consider subordinate perception.	Pilot fails to consider flight engineer perception.	Surgeon fails to consider nurse perception.
	10	Subordinate fails to communicate problem.	Flight engineer fails to communicate a low fuel state	Nurse fails to communicate patient breathing state
	11	Leader and Subordinate have incorrect perception, but with communication.	Pilot and flight engineer have incorrect perception, but with communication.	Surgeon and Nurse have incorrect perception, but with communication.
	12	A problem exists with incorrect perception by the leader and subordinate, and incorrect action	A problem exists with incorrect perception by the pilot and flight engineer, and incorrect action	A problem exists with incorrect perception by the surgeon and nurse, and incorrect action
Warnings	13	Non-communication from a subordinate precedes leader reception of communication from a different subordinate	Non-communication from an engineer precedes pilot reception of communication from another engineer	Non-communication from a nurse precedes surgeon reception of communication from a different nurse
	14	Non-communication from a subordinate with no interaction to the leader	Non-communication from an engineer with no interaction to the leader	Non-communication from a nurse with no interaction to the surgeon

4.3.1.6.3 Probability of each trace: As detailed in Appendix B, deriving the probability of each trace becomes possible by implementing a Bayesian belief network that is derived from the behavior model. The constraints change the inherent structure of the belief network, and so care must be taken to ensure consistency between the behavior model and the belief network.

Figures 4.48 and 4.49 illustrate that the model topology and number of alternatives for each event produce a Bayesian belief network such that the probability of each event is dependent upon the alternatives that have occurred prior to that event.

Table 4.6 shows the results after running the MATLAB[®] script that aligns the probability belief network with the specific constraints applied to this model. The derivation and results of the script are also listed in Appendix B. Figure 4.7 illustrates how the developer can set performance goals using the belief network.

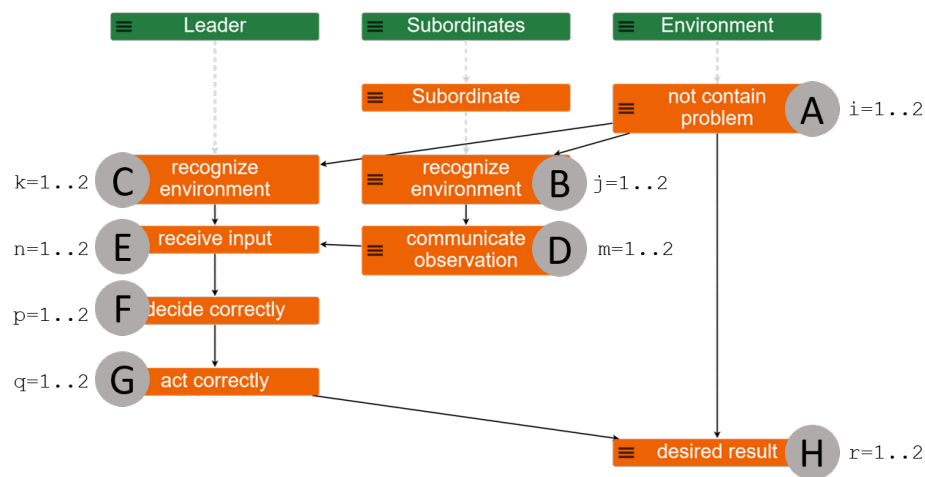


Figure 4.48. Decision Model alternative events for each event, executed at *scope* = 1, indicating that the model had one subordinate.

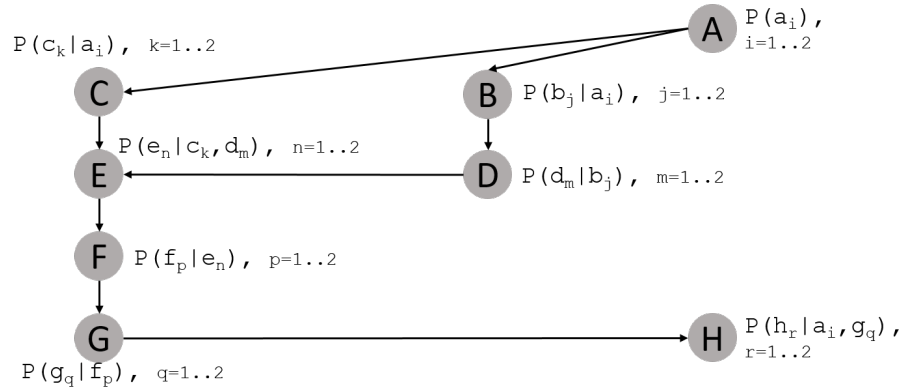


Figure 4.49. Decision Model probability network derived from the model topology and number of alternatives of each event.

Table 4.6. Decision Model probability calculations, given a probability of 0.5 for each alternative event, executed at *scope* = 1, indicating that the model had one subordinate. Trace 9 is highlighted since that pattern matches the narrative of the aviation mishap and surgical crisis.

Trace	Ref	P(trace)	i i = 1: not contain problem i = 2: contain problem	P(a)	j j = 1: Subordinate recognizes environment j = 2: Subordinate does not recognize environment	P(b)	k k = 1: Leader recognizes environment k = 2: Leader does not recognize environment	P(c)	m m = 1: communicate observation m = 2: not communicate observation	P(d)	n n = 1: receive input n = 2: not receive input	P(e)	p p = 1: decide correctly p = 2: not decide correctly	P(f)	q q = 1: act correctly q = 2: not act correctly	P(g)	r r = 1: desired result r = 2: not desired result	P(h)	Template
1	1	0.125	1	0.50	1	0.50	1	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 1: No problem
2	129	0.125	2	0.50	1	0.50	1	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 2: Problem exists, correct perception
3	65	0.125	1	0.50	2	0.50	1	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 3: No problem, incorrect subordinate
4	193	0.125	2	0.50	2	0.50	1	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 4: Problem exists, incorrect subordinate
5	33	0.125	1	0.50	1	0.50	2	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 5: No problem, incorrect leader
6	161	0.03125	2	0.50	1	0.50	2	0.50	1	0.50	1	0.50	1	1.00	1	1.00	1	1.00	Template 6: Problem, incorrect leader perception
7	97	0.125	1	0.50	2	0.50	2	0.50	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 7: No problem, incorrect perception
8	225	0.03125	2	0.50	2	0.50	2	0.50	1	0.50	1	0.50	1	1.00	1	1.00	1	1.00	Template 8: Problem, incorrect perception
9	176	0.03125	2	0.50	1	0.50	2	0.50	1	0.50	2	0.50	2	1.00	2	1.00	2	1.00	Template 9: Leader fails to consider subordinate
10	192	0.0625	2	0.50	1	0.50	2	0.50	2	0.50	2	1.00	2	1.00	2	1.00	2	1.00	Template 10: Problem, subordinate fails to communicate
11	240	0.03125	2	0.50	2	0.50	2	0.50	1	0.50	2	0.50	2	1.00	2	1.00	2	1.00	Template 11: Problem, comm incorrect perception
12	256	0.0625	2	0.50	2	0.50	2	0.50	2	0.50	2	1.00	2	1.00	2	1.00	2	1.00	Template 12: Problem, all went wrong

Probability of success:

$$\sum_{trace=1}^8 P_{trace} = 0.8125$$

Probability of failure:

$$\sum_{trace=9}^{12} P_{trace} = 0.1875$$

The developer may use this belief network to investigate areas where improved performance would give better results. Considering the following assumptions and goals, the analysis was executed again as follows:

Assumptions and goals for improved performance include the following:

- A problem exists only 20% of the time,
- The leader has correct perception 70% of the time,
- The subordinate has correct perception 70% of the time,
- If the subordinate has an accurate perception, then the subordinate communicates 80% of the time
- If the subordinate communicates, then the leader receives the communication 70% of the time.

Probability of success:

$$\sum_{trace=1}^8 P_{trace} = 0.97$$

Probability of failure:

$$\sum_{trace=9}^{12} P_{trace} = 0.03$$

Table 4.7. Decision Model probability after attributing goals for the probability belief network.

			i = 1: not contain problem i = 2: contain problem		j = 1: Subordinate recognizes environment j = 2: Subordinate does not recognize environment		k = 1: Leader recognizes environment k = 2: Leader does not recognize environment		m = 1: communicate observation m = 2: not communicate observation		n = 1: receive input n = 2: not receive input		p = 1: decide correctly p = 2: not decide correctly		q = 1: act correctly q = 2: not act correctly		r = 1: desired result r = 2: not desired result		
Trace	Ref	P(trace)	i	P(a)	j	P(b)	k	P(c)	m	P(d)	n	P(e)	p	P(f)	q	P(g)	r	P(h)	Template
1	1	0.392	1	0.80	1	0.70	1	0.70	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 1: No problem
2	129	0.098	2	0.20	1	0.70	1	0.70	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 2: Problem exists, correct perception
3	65	0.168	1	0.80	2	0.30	1	0.70	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 3: No problem, incorrect subordinate
4	193	0.042	2	0.20	2	0.30	1	0.70	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 4: Problem exists, incorrect subordinate
5	33	0.168	1	0.80	1	0.70	2	0.30	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 5: No problem, incorrect leader
6	161	0.02352	2	0.20	1	0.70	2	0.30	1	0.80	1	0.70	1	1.00	1	1.00	1	1.00	Template 6: Problem, incorrect leader perception
7	97	0.072	1	0.80	2	0.30	2	0.30	1	1.00	1	1.00	1	1.00	1	1.00	1	1.00	Template 7: No problem, incorrect perception
8	225	0.0063	2	0.20	2	0.30	2	0.30	1	0.50	1	0.70	1	1.00	1	1.00	1	1.00	Template 8: Problem, incorrect perception
9	176	0.01008	2	0.20	1	0.70	2	0.30	1	0.80	2	0.30	2	1.00	2	1.00	2	1.00	Template 9: Leader fails to consider subordinate
10	192	0.0084	2	0.20	1	0.70	2	0.30	2	0.20	2	1.00	2	1.00	2	1.00	2	1.00	Template 10: Problem, subordinate fails to communicate
11	240	0.0027	2	0.20	2	0.30	2	0.30	1	0.50	2	0.30	2	1.00	2	1.00	2	1.00	Template 11: Problem, comm incorrect perception
12	256	0.009	2	0.20	2	0.30	2	0.30	2	0.50	2	1.00	2	1.00	2	1.00	2	1.00	Template 12: Problem, all went wrong

4.3.1.6.4 The Design Structure Matrix (DSM): This analysis element captures the sum of all interactions among events for all traces produced by executing MP. Each trace of the model is a directed graph without loops, termed a Directed Acyclic Graph (DAG). The automated script described in Appendix C tabulates the output of the model. The results for scope one through three are shown in Tables 4.8 through 4.10, respectively. These graphs tabulate the number of relationships found in the model, listed event by event. For example the event “communicate observation” precedes “recognize environment” and Table 4.8 indicates that this occurred a total of eight times for all traces at a scope of execution of one. Tables 4.9 and 4.10 shows these results at the scope of execution of two and three, respectively.

In general, analysis techniques may readily be applied to Directed Acyclic Graphs (DAGs). Comparison of results and clustering analysis were demonstrated through the results shown in Appendix C and summarized below:

Comparison at increased scope: The developer may need to know whether the increase in scope of execution results in any new relationships or interactions in the model. A simple comparison of the number of unique relationships of the model was conducted analytically by tabulating the

number of unique edges in the model. Since the results for comparing scope one to two and one to three are consistent, the developer may be assured that increasing the scope provides no new relationships for this model. Appendix C shows the derivation of this comparison using MATLAB®.

Comparison of results from scope of one to two:

```
Sum of DSM unique edges from Decision Model Scope 1 is 58  
Sum of DSM unique edges from Decision Model Scope 2 is 58  
All relationships between these models are consistent.
```

Comparison of results from scope of one to three:

```
Sum of DSM unique edges from Decision Model Scope 1 is 58  
Sum of DSM unique edges from Decision Model Scope 3 is 58  
All relationships between these models are consistent.
```

Table 4.8. The Decision Model DSM, scope of 1.

Event name	To	Leader	Subordinates	Environment	recognizeenvironment	receiveinput	decidecorrectly	actcorrectly	Subordinate	communicateobservation	notcontainproblem	desiredresult	containproblem	notrecognizeenvironment	notreceiveinput	notdecidecorrectly	notactcorrectly	notdesiredresult	notcommunicateobservation
From		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Leader	(1)				4	8	8	8						8	4	4	4		
Subordinates	(2)								12										
Environment	(3)										4	8	8					4	
recognizeenvironment	(4)					4				5									1
receiveinput	(5)						8												
decidecorrectly	(6)							8											
actcorrectly	(7)											8							
Subordinate	(8)				6					10				6					2
communicateobservation	(9)					8									2				
notcontainproblem	(10)				4							4		4					
desiredresult	(11)																		
containproblem	(12)				6							4		10				4	
notrecognizeenvironment	(13)					4				5					4				1
notreceiveinput	(14)															4			
notdecidecorrectly	(15)																4		
notactcorrectly	(16)																	4	
notdesiredresult	(17)																		
notcommunicateobservation	(18)														2				

Table 4.9. The Decision Model DSM, scope of 2.

Event name	To	Leader	Subordinates	Environment	recognizeenvironment	receiveinput	decidecorrectly	actcorrectly	Subordinate	communicateobservation	notcontainproblem	desiredresult	containproblem	notrecognizeenvironment	notreceiveinput	notdecidecorrectly	notactcorrectly	notdesiredresult	notcommunicateobservation
From		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Leader	(1)				24	48	48	48						40	16	16	16		
Subordinates	(2)								128										
Environment	(3)										24	48	40					16	
recognizeenvironment	(4)					24				40									24
receiveinput	(5)						48												
decidecorrectly	(6)							48											
actcorrectly	(7)											48							
Subordinate	(8)				64					80				64					48
communicateobservation	(9)					64									28				
notcontainproblem	(10)				36							24		36					
desiredresult	(11)																		
containproblem	(12)				52							24		68				16	
notrecognizeenvironment	(13)					24				40					16				24
notreceiveinput	(14)															16			
notdecidecorrectly	(15)																16		
notactcorrectly	(16)																	16	
notdesiredresult	(17)																		
notcommunicateobservation	(18)														16				

Table 4.10. The Decision Model DSM, scope of 3.

Event name	To	Leader	Subordinates	Environment	recognizeenvironment	receiveinput	decidecorrectly	actcorrectly	Subordinate	communicateobservation	notcontainproblem	desiredresult	containproblem	notrecognizeenvironment	notreceiveinput	notdecidecorrectly	notactcorrectly	notdesiredresult	notcommunicateobservation
From		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Leader	(1)				112	224	224	224						176	64	64	64		
Subordinates	(2)								864										
Environment	(3)										112	224	176					64	
recognizeenvironment	(4)					112				240									192
receiveinput	(5)						224												
decidecorrectly	(6)							224											
actcorrectly	(7)											224							
Subordinate	(8)				432					480				432					384
communicateobservation	(9)					384								184					
notcontainproblem	(10)				224							112		224					
desiredresult	(11)																		
containproblem	(12)				320							112		384				64	
notrecognizeenvironment	(13)					112				240					64				192
notreceiveinput	(14)															64			
notdecidecorrectly	(15)																64		
notactcorrectly	(16)																	64	
notdesiredresult	(17)																		
notcommunicateobservation	(18)														96				

Cluster Analysis of the DSM: Figure 4.50 illustrates the utility of the clustering algorithm, as applied to trace 10 of the Decision Model.

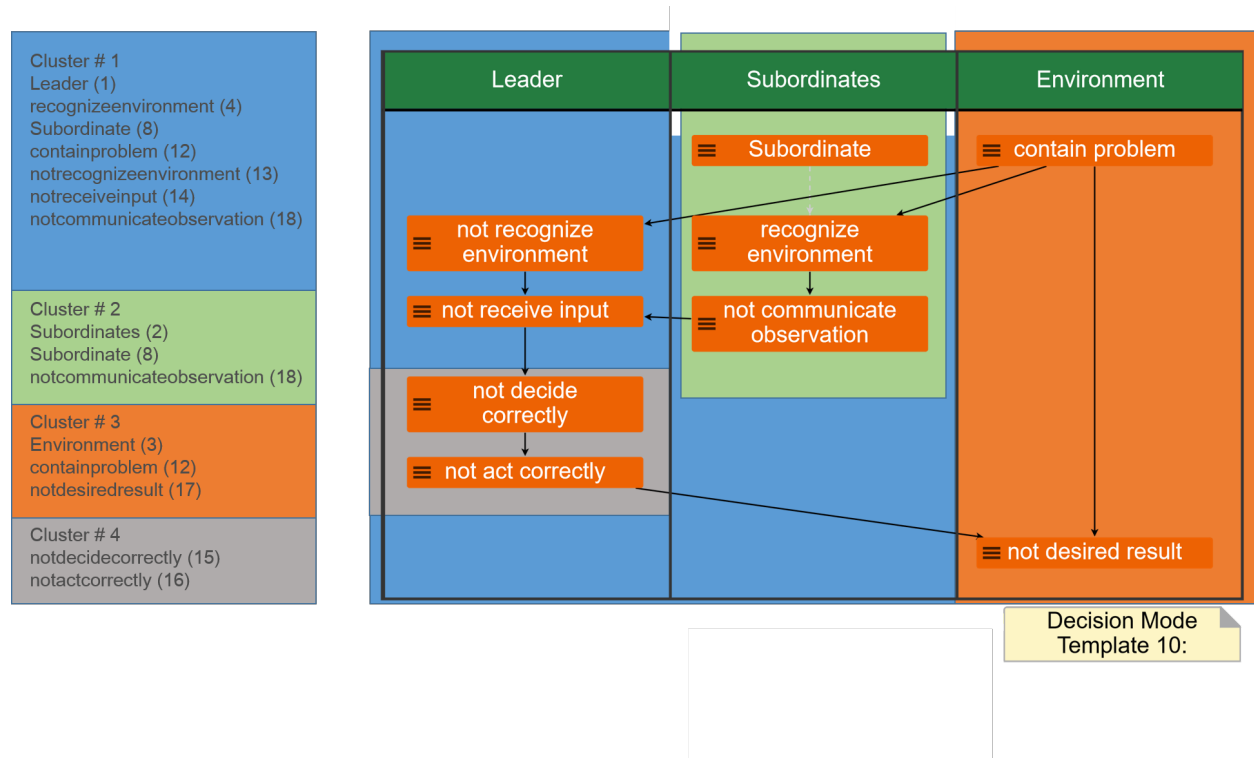


Figure 4.50. Decision Model cluster algorithm results.

4.3.1.7 Decision Model: Model re-use

The following paragraphs describe reuse applications for the surgical model and aviation model. The surgical model applies a word for word replacement from the abstraction-level, written in a natural way, consistent with the domain. The aviation model applies the corresponding word for word changes and also adds intermediate events, illustrating that the identical templates still capture the patterns of interest to the developer.

4.3.1.7.1 The Surgical Decision Model: As depicted in Figure 4.51, the model was implemented by a word-for-word replacement of the events from the abstraction-level model with the surgical model events. The identical model-specific and universal templates were applied from the repository, and all templates were demonstrated to be effective in the applied surgical form of the model.

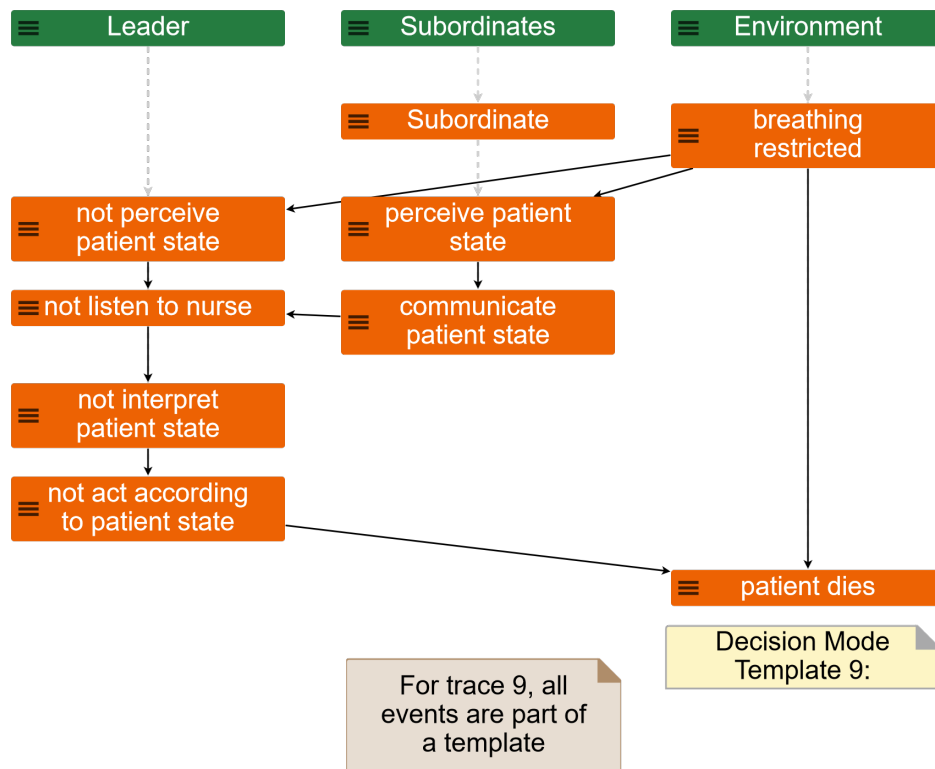


Figure 4.51. Decision Model surgical application, illustrating Template 9.

4.3.1.7.2 The Aviation Decision Model: This model adds a bit more complexity, as illustrated in Figure 4.52. For this model, additional alternate events were added to the behavior descriptions of the environment (aircraft state), leader (pilot), and subordinate (flight engineer). These alternative events are denoted by those encased by asterisks in the following segments of code. This notation enables zero to many, up to the scope of execution. Adding these six alternative events, increased the number of traces at scope=1, from twelve to 1536.

MP code segment 4.31: Alternative events added to the aircraft state behavior.

```

ROOT Aircraft_state:
    (* normal_initial_flight *)
    ( not_low_fuel
      | low_fuel )
    ( safe_landing
      | crash );

```

In practice, the developer may associate these non-essential events with an additional attribute, and then ignore those occurrences during the template query. Upon execution of the model, all templates derived the associated patterns of behavior, ignoring the alternatives of events that were

MP code segment 4.32: Alternative events added to the pilot behavior.

```

ROOT Pilot:
    (* experience_long_flight_duration *)
    (* observe_landing_gear_fault_indication *)
    ( perceive_fuel_level
      | not_perceive_fuel_level )
    (* consider_multiple_fault_indications *)
    ( listen_to_engineer
      | not_listen_to_engineer )
    (* experience_distraction *)
    (* delay_decision *)
    ( interpret_aircraft_state
      | not_interpret_aircraft_state )
    ( act_according_to_fuel_state
      | not_act_according_to_fuel_state );

```

MP code segment 4.33: Alternative events added to the flight engineer behavior.

```

ROOT Flight_engineers: {+ <1> Flight_engineer +};
Flight_engineer:
    (* maintain_aircraft_status *)
    ( perceive_fuel_level
      | not_perceive_fuel_level )
    ( communicate_fuel_level
      | not_communicate_fuel_level );

```

not essential to the pattern. An example of the associated code and the adjustment to the universal template search is listed as follows:

MP code segment 4.34: Designation of an event as non-essential to the template.

```

COORDINATE $x:experience_long_flight_duration
DO $x.nEssential := true; OD;

```

MP code segment 4.35: Adjusting the universal template to ignore non-essential events.

```

/*Mark the traces that have essential events not in a template*/
/**/
COORDINATE      $a:$EVENT SUCH THAT ( NOT $a.inTemplate AND
                                         NOT $a.nEssential )
DO ADD SAY("not_in_template") not_in_template $a;
GLOBAL.count += 1;
count += 1;
MARK;
OD;

```

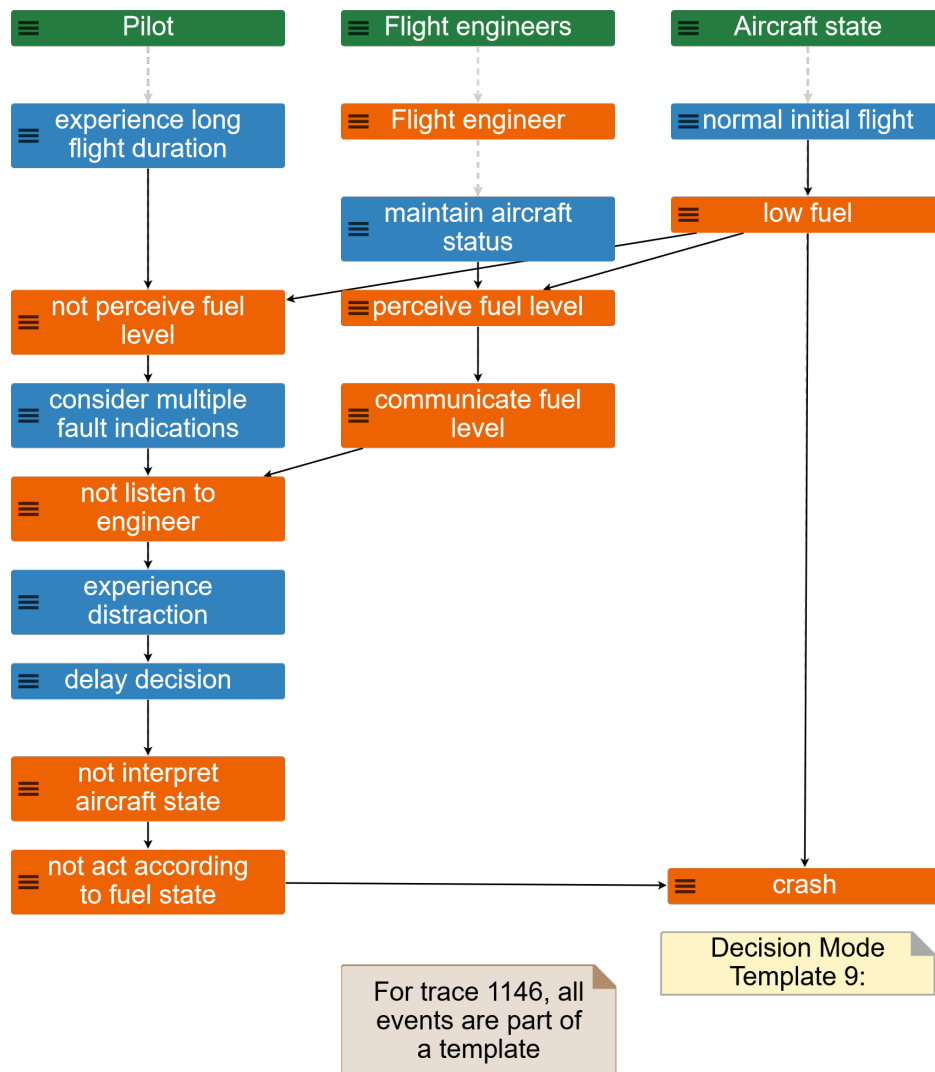


Figure 4.52. Decision Model aviation mishap application, illustrating trace 1146 of 1536.

Key points:

1. Specific-model templates capture a range of problems as a word-for-word replacement based on the abstracted model.
2. More complex models may add events for improved context, but these events are not essential to the defined set templates.
3. The set of universal templates identify essential events that are not part of an existing template, alerting the developer to consider the newly identified pattern and decide whether to add constraints or an additional template to capture that pattern.

4.3.1.8 Implications of the Decision Model

The Decision Model and the associated probability belief network provides insight into where changes can be made to the system, and a mathematical construct determining the impact of changes. Syed's main point in analyzing the failure method, was captured in Figure 4.51, where the leader fails to consider the input of a subordinate, then resulting in a failed and serious outcome [17]. The objective of the case study was to illustrate that the aviation community immediately responded by conducting an independent analysis of the failure, communicating these results to the aviation community, and instituting changes in training. These responses encouraged the subordinates to have effective communication in an emergency and changed the leader's willingness to receive information during the crisis. Conversely, the medical community has a history of not disclosing the details of a failed outcome, which leads to a greater likelihood of not changing the probabilities associated with patterns of behavior.

4.3.2 Derivation of the Object-to-Effect Model

The Object-to-Effect Model reflects an operational task that relies upon one to multiple sensors to produce compiled information that is sufficient for a given critical action, thereby having an effect on an object of interest. The model was derived from concepts described by Dahmann et al. [18] and Antul et al. [19] that modeled an operations center response to a civilian aircraft entering a restricted zone. The Object-to-Effect Model demonstrates the following:

- Uses Monterey Phoenix (MP) to model distributed roles including a sensors, processor (i.e., the operations center), an actor, and objective.
- Verifies the methodology, outlined in Section 4.1, using a simple MP example, indicating criteria to repeat a step, to move forward, or to step backward.
- Employs constraints to shape the system outcomes, types of constraints include logical, definition, simplification, and design.
- Incorporates interactions and constraints to formally define system behavior requirements.
- Identifies inherent patterns of the model by deriving templates based on the event attributes and model topology. Subsequent outcomes, executed at an increased scope of execution consist of combinations of these templates.
- Employs universal templates to identify pattern segments, including the successful and failed sequences, warnings, and the OODA loop.
- Verifies the method of pattern identification, based on the architecture structure (i.e., topology), combined with event semantics.
- Verifies the effects of increased scope such that all results at an increased scope are combinations of templates derived at lower scope of execution. Specifically, twenty-two templates were derived, and the results at higher scope were shown to be combinations of these templates.
- Verifies that all events of all traces are part of a template, thereby indicating that the developer has insight to all possible combinations of results.
- Uses the behavior model to derive a Bayesian belief network.
- Derives the probability of occurrence of each trace of the model, demonstrated at scope of one object of interest, one sensor, one processor, and one actor.
- Implements an automated algorithm to trim branches of the probability belief network, based on the constraints.
- Implements an automated algorithm to derive a Design Structure Matrix (DSM) that enables the developer to investigate both frequent and rare occurrences of patterns of behavior.
- Illustrates reuse of a SoS schema by re-organizing new combinations of the functions asso-

ciated with the sensor, processors, and actors.

- Illustrates reuse of the model for a vehicle control application.

Derivation of the Object-to-Effect Model follows the same approach as defined by the Decision Model, outlined in the previous section. Simulating the use of a repository, as illustrated in Figure 4.53, two categories of templates are employed: 1) model-type specific templates and 2) universal templates. In this instance, the type-specific templates were derived specifically to model an effect upon some object of interest. The universal templates derived from the Decision Model directly apply to the Object-to-Effect Model.

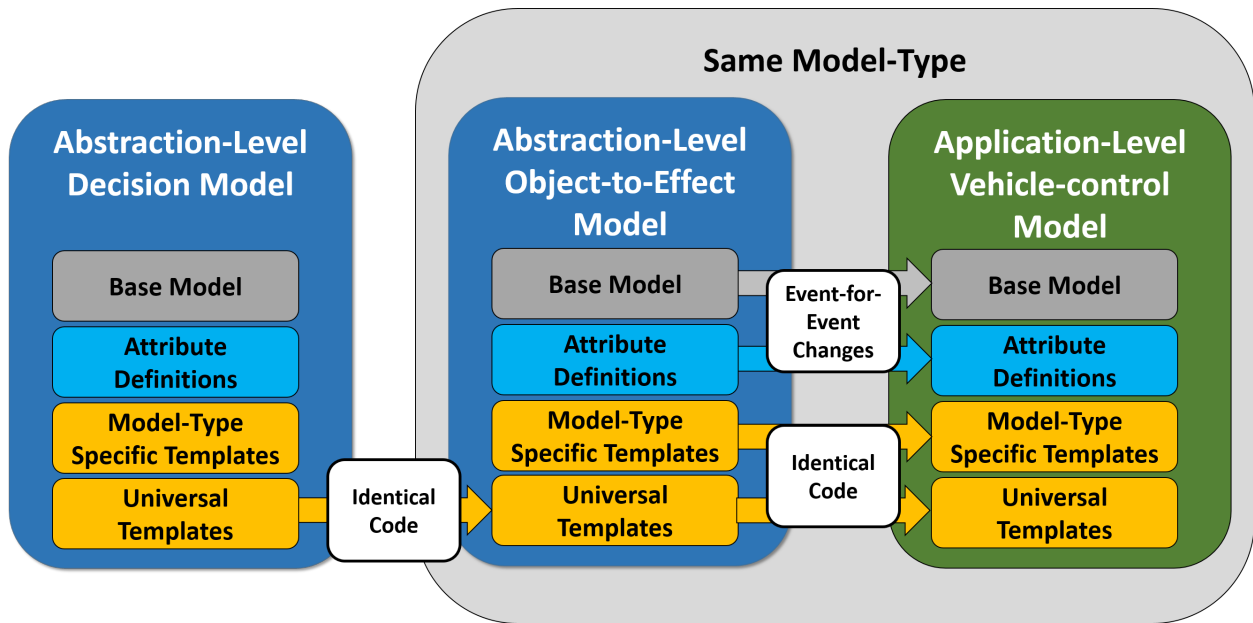


Figure 4.53. Universal and model-type specific templates used identically across the abstraction-level and application-level models.

4.3.2.1 Object to Effect Model: Behavior narrative

Inspired by the publications by Dahmann et al. [18] and Antul et al. [19], distributed operations to address an aircraft that has entered a restricted zone without authorization. The relevant behaviors of the model include the events associated with the environment, sensor(s), operations center(s), actors(s), and the object(s) of interest. The construct of this behavior model enables adaptive interactions among each of these events so that the objectives might be attained within a changing and complex environment. The model was developed in order to allow multiple sets of sensors, operation centers (abstracted as processors), response aircraft (abstracted as actors), and the unauthorized aircraft (abstracted as objects of interest). Executing the model at a scope of one, so that

just one of each of these systems are represented, resulted in 12,228 traces as illustrated in Appendix A.2. Increasing the scope of execution is enabled by increasing any of the individual entities. For example, the scope annotated as (2-1-1-1) refers to two objects, one sensor, one processor, and one actor. The run matrix of this model references total potential outcomes up to the tens of millions, with valid traces in the thousands, as indicated in Section 4.3.2.6. By implementing model-specific and universal templates, this model enables the developer to assess all possible outcomes up to the scope of execution.

Figure 4.54 illustrates one of the traces produced by the model, with an object of interest receiving the desired effect. The next sections outline each step of the methodology employed to develop the behavior model illustrated in this figure.

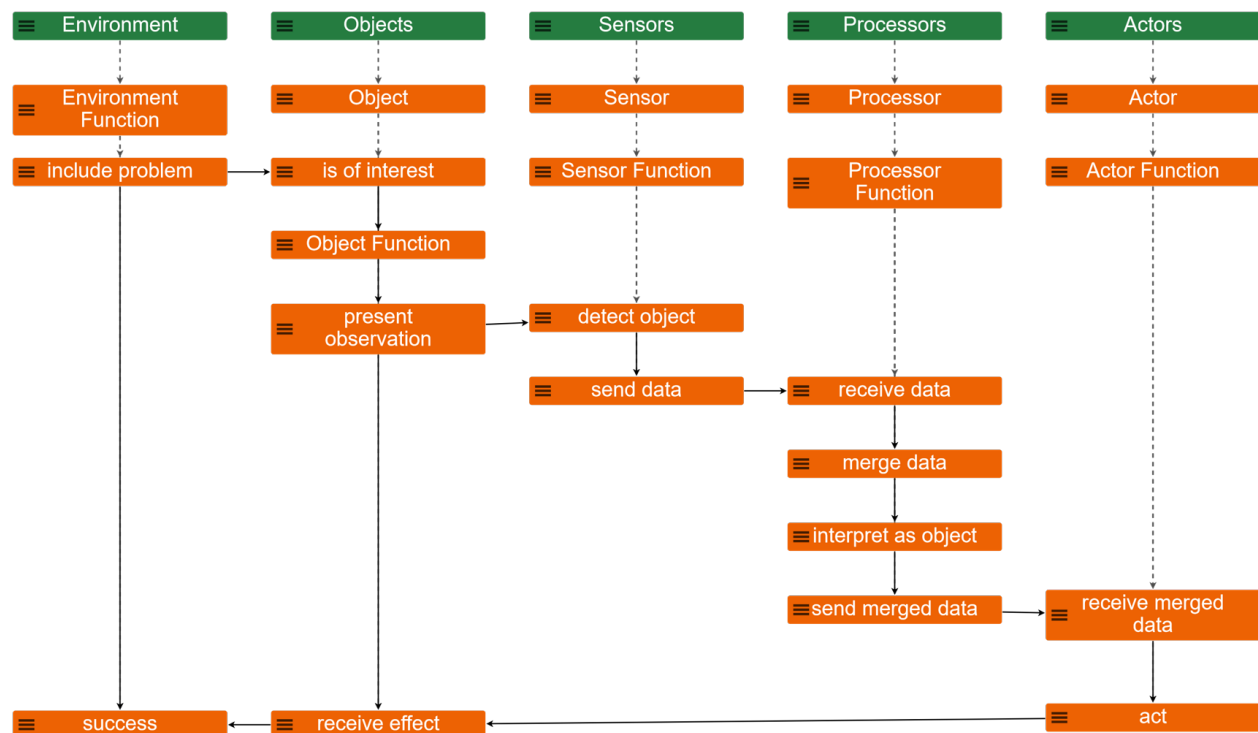


Figure 4.54. Object-to-Effect Model, reflecting trace 9 such that an object of interest receives the desired effect, illustrated at scope of (1-1-1-1) with one object, one sensor, one processor, and one actor.

4.3.2.2 Object-to-Effect Model: Event identification

As with the Decision Model, there exists an abstraction-level description of the model, while the application-level models may include any of the following:

- Operations center response to an aircraft entering a restricted area without authorization,
- Rescue center response to finding victims in a natural disaster,
- Networked traffic and vehicle control systems, and
- Home security system.

The terms used at an abstract layer may become a bit difficult to follow, and so it is often beneficial to understand the logic behind one of the applications. For example, the user may more easily understand the processor interpretation of a hostile, friendly, or unknown, as opposed to the interpretation of an object of interest, not an object of interest, or unknown of the Abstraction-level model. However, considering the abstraction-level model enables a wider application of the search templates. These templates operate on the attributes described at the abstraction-level model as opposed to the specific application. In this way, the identical templates may be used for any appropriate application that include objects of interest, sensors to identify these objects, processors that collated the data, and actors that respond to the objects of interest. The developer simply applies abstracted-level event attributes the natural language of the domain of a specific model.

4.3.2.2.1 Environment events: As illustrated in Figure 4.55, these events include those associated with the environment function, consisting of the existence of a problem and the resulting successful or failed outcome. Only one instance of the environment function exists for any scope of execution of the model.

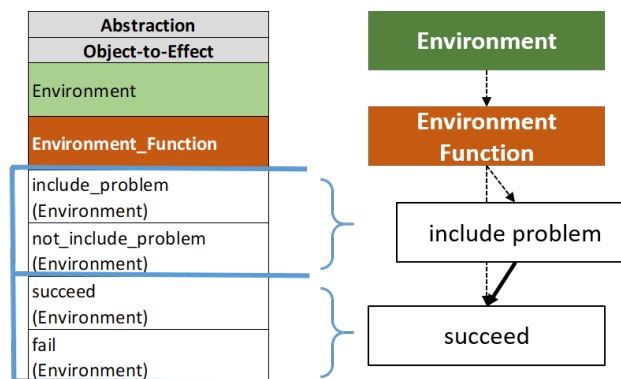


Figure 4.55. Object-to-Effect Model environment events are composed within the environment function, consisting of whether a problem exists followed by whether the outcome succeeds or fails. The existence of a problem precedes the outcome, as indicated by the solid arrows. Each of these events are included in the function, as indicated by the dashed-line arrows.

MP code for the set of environment events: This code includes the root definition of the environment of the environment function. The environment function has two sets of alternative events: (1) whether to include a problem, and (2) whether the outcome succeeds or fails. The inclusion of the problem precedes the success or failure of the system.

MP code segment 4.36: Environment events.

```

/****Environment*****/
ROOT Environment: { Environment_Function }
Environment_Function:
  ( not_include_problem | include_problem )
  ( succeed | fail )

```

4.3.2.2.2 Object events: As illustrated in Figures 4.56 and 4.57, these events include the behaviors of the a single objects or multiple objects, respectively. Each object may be of interest or *not* of interest. The object functions include presentation of an observation and then receiving an effect. Alternative events for each object include whether the object is of interest, whether the object presents an observation, and whether an effect acts upon the object.

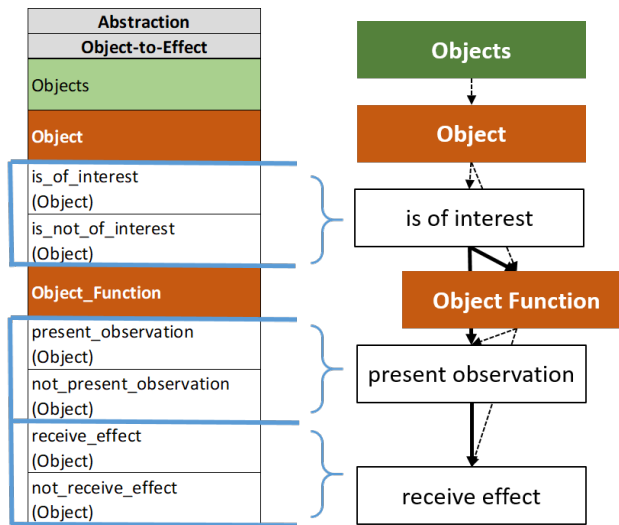


Figure 4.56. Object-to-Effect Model object events indicate whether the object is of interest to the system, followed by the functions of presenting an observation and receiving the effect from the system acting upon it. The object identity precedes the object function, and presenting an observation precedes the effect, as designated by the solid-line arrows.

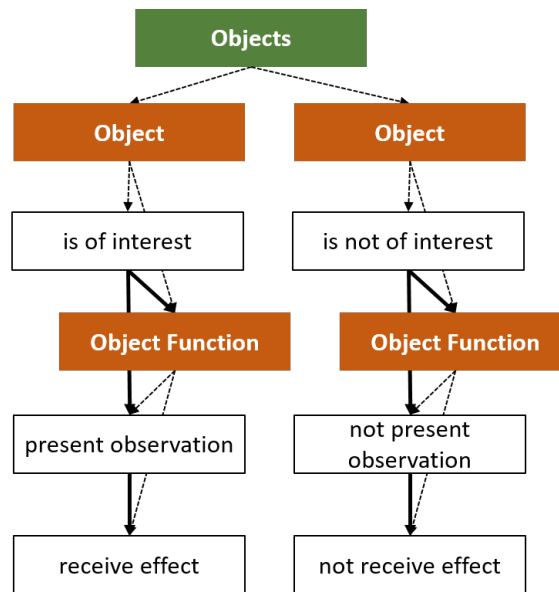


Figure 4.57. Object-to-Effect Model multiple objects, each may or may not be of interest to the system, and each object has a separate execution of the alternatives of events.

The MP code for the set of object events: This code includes the root of "Objects" followed by one or more object. Each object may be of interest to the system, and each object also has a separate instance of the object function. The object function includes whether to present the observation and whether to receive the effect from the system. Increasing the number of objects is changed by adjusting the number within the root definition, (i.e., change < 1 > to < 2 > for two objects in the model). The following lists the code for this segment.

MP code segment 4.37: Object events.

```

/****Objects*****/
ROOT Objects: { + <1> Object + }
Object: ( is_not_of_interest | is_of_interest )
         Object_Function
Object_Function:
( present_observation | not_present_observation)
( receive_effect      | not_receive_effect      )

```

4.3.2.2.3 Sensor events: As illustrated in Figures 4.58 and 4.59, these events include the behaviors of the a single sensor or multiple sensors, respectively. The sensor functions include detecting the object and sending data to the processor(s).

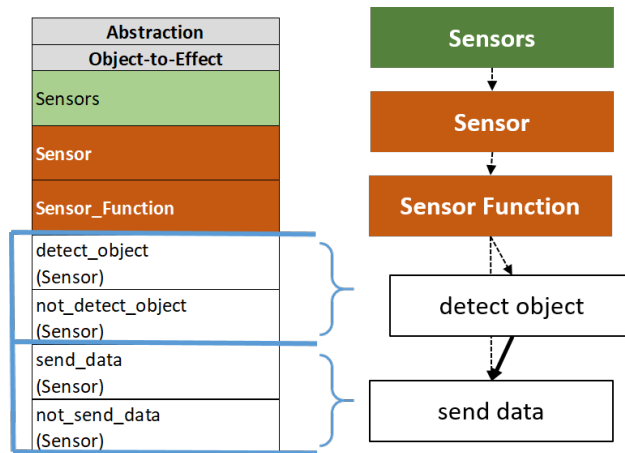


Figure 4.58. Object-to-Effect Model sensor events are composed within the Sensor Function, and include whether the sensor detects the object and whether the data is sent to the processor. Detecting the object precedes sending the data, as indicated by the solid-line arrow. Each of these events are included in the sensor function, as indicated by the dashed-line arrow.

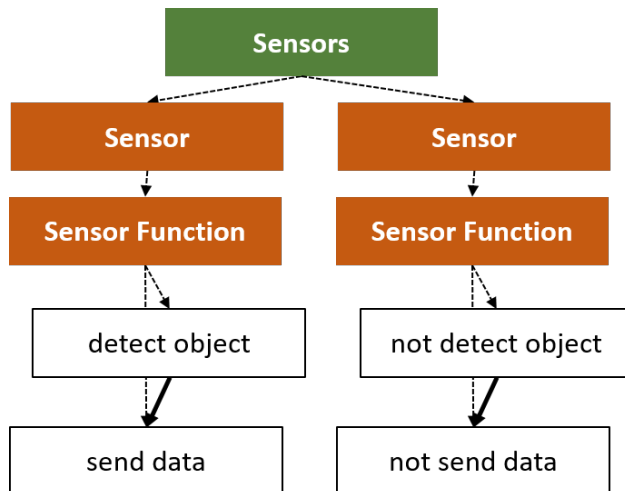


Figure 4.59. Object-to-Effect Model multiple sensor events include all combinations, an example of two sensors are illustrated in this diagram.

MP code for the set of sensor events: This code includes the root of "Sensors" followed by one or more sensors. Each sensor has a set of functions, each with two alternatives, defined within the schema as follows:

MP code segment 4.38: Sensor events.

```

ROOT Sensors:      { + <1> Sensor      + }
Sensor: Sensor_Function;
Sensor_Function:
    ( detect_object | not_detect_object )
    ( send_data     | not_send_data     );

```

4.3.2.2.4 Processor events: As illustrated in Figures 4.60 and 4.61, these events include the behaviors of the a single sensor or multiple sensors, respectively. The processor functions include receiving data from the sensor(s), merging the data, interpreting the data, and sending the merged solution. The interpretation includes three options: the object of interest, not an object of interest, or unknown.

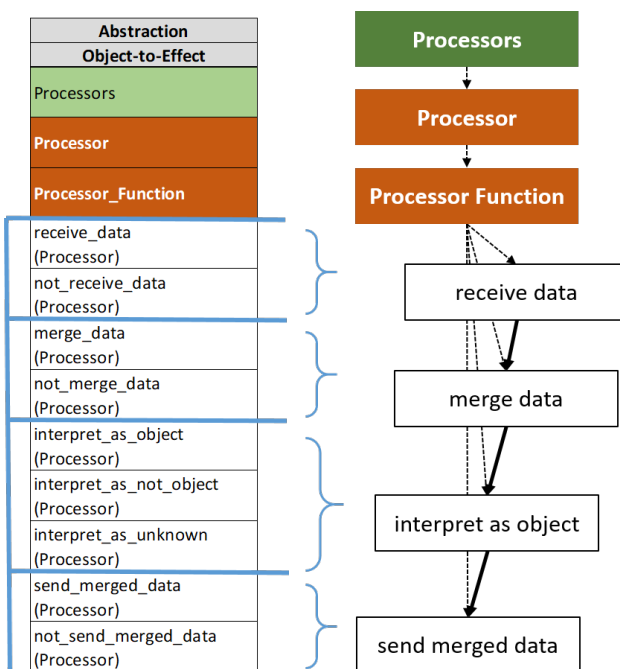


Figure 4.60. Object-to-Effect Model processor events are composed within the processor function consisting of whether data is received, merged, interpreted, and sent to the actor, as indicated by the dashed-line arrows. Each of these functions precede the next, as indicated by the solid-line arrows.

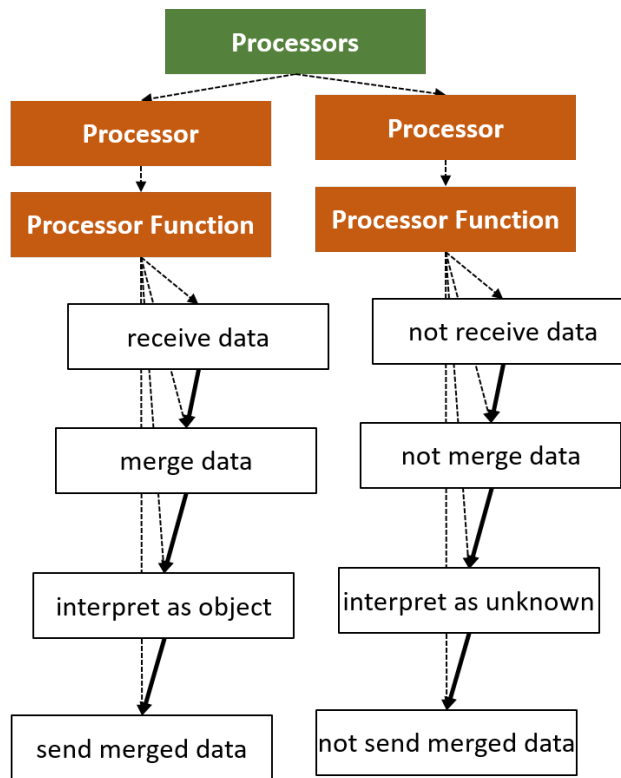


Figure 4.61. Object-to-Effect Model multiple processors may be employed, as illustrated by this example.

MP code for the set of processor events: This code includes the root of "Processors" and one to many individual processors. Each processor independently performs a function that consists of receiving data from the sensor, merging data, interpreting the data, and sending the merged data to the actor.

MP code segment 4.39: Processor events.

```

ROOT Processors: { + <1> Processor + }
Processor: Processor_Function;
Processor_Function:
  ( receive_data          | not_receive_data      )
  ( merge_data           | not_merge_data       )
  ( interpret_as_not_object | interpret_as_object  )
  ( interpret_as_unknown  | interpret_as_unknown )
  ( send_merged_data      | not_send_merged_data );
  
```

4.3.2.2.5 Actor events: As illustrated in Figures 4.62 and 4.63, these events include the behaviors of a single actor or multiple actors, respectively. The actor functions include receiving merged data

and then acting according to the data available. The merged data identifies whether the object is of interest, not of interest, or unknown.

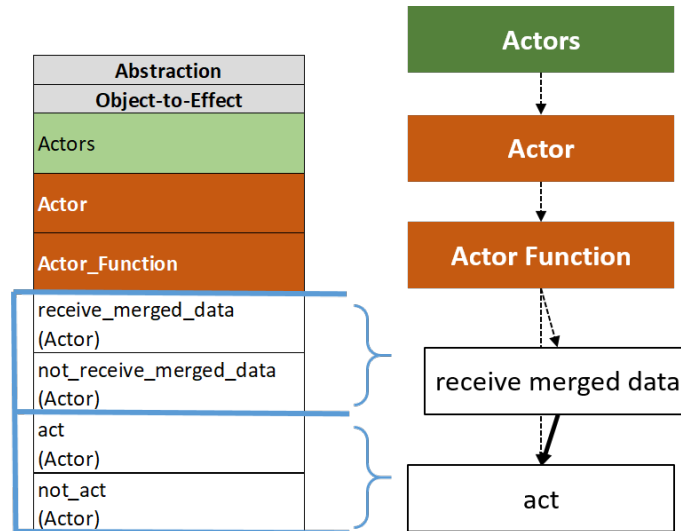


Figure 4.62. Object-to-Effect Model actor events are composed within the actor function and consist of receiving the merged data from the processor and acting upon those data in order to have an effect on the object of interest. Receiving the data precedes the action, as indicated by the solid-line arrow. The actor functions consist of receiving the data and taking an action, as indicated by the dashed-line arrows.

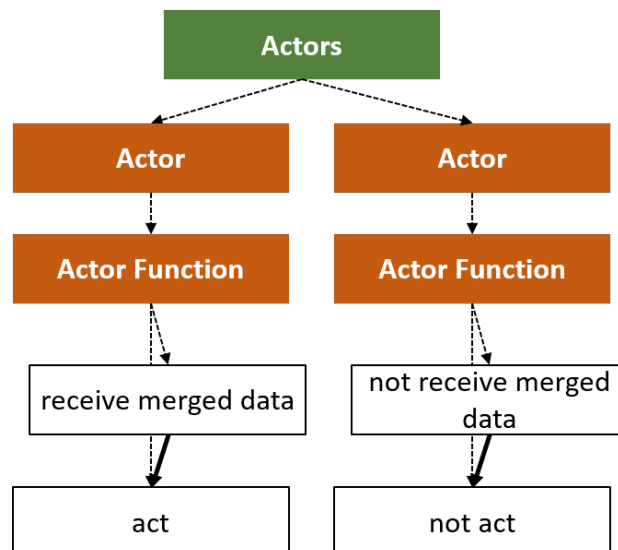


Figure 4.63. Object-to-Effect Model multiple actors.

MP code for the set of actor events: This code includes the root of "Actors" followed by one or many actors. Each actor has a set of functions, defined within the schema as follows:

MP code segment 4.40: Actor events.

```

ROOT Actors:      { + <1> Actor      + }
Actor: Actor_Function;
Actor_Function:
  ( receive_merged_data | not_receive_merged_data )
  ( act                 | not_act                 );

```

4.3.2.3 Object to Effect Model: Coordination definition

Coordination defines the interactions of the environment, the set objects, the set of sensors, the set of processors, and the set of actors. Graphically, the coordination crosses the swim-lane boundaries of each of these sets of root events.

4.3.2.3.1 Coordination of environment to and from objects: As illustrated in Figure 4.64 and Table 4.11 this coordination includes two interactions: (1) a problem precedes the existence of an object of interest, and (2) the effect of an action determines success or failure.

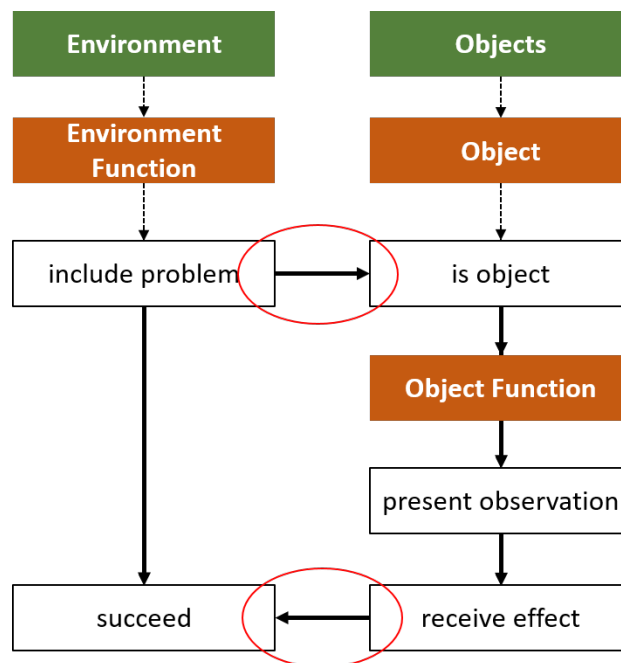


Figure 4.64. Object-to-Effect Model interaction of the environment with the objects.

Table 4.11. Object-to-Effect Model interaction of the environment with the objects. The existence of a problem leads to the alternatives of whether the object is of interest and whether the effect upon the object leads to a successful outcome.

Ref	Abstraction-Level
	Object-To-Effect
1	Existence of a problem precedes existence of the object
2	The effect of the action determines the outcome of success or failure

MP code for the coordination of the environment and objects: The code is listed as follows:

MP code segment 4.41: Coordination of the environment with the objects.

```

/* Interaction #1: Problem state precedes object identity */
COORDINATE $p: (not_include_problem | include_problem)
DO COORDINATE $t: (is_of_interest | is_not_of_interest)
DO ADD $p PRECEDES $t; OD; OD;

/* Interaction #2: The effect precedes outcome state */
COORDINATE $h: (receive_effect | not_receive_effect)
DO COORDINATE $s: (succeed | fail)
DO ADD $h PRECEDES $s; OD; OD;

```

4.3.2.3.2 Coordination of objects to sensors: As illustrated in Figure 4.65 and Table 4.12 the coordination of objects to sensors includes only one interaction: the object presentation precedes the sensor detection of the object.

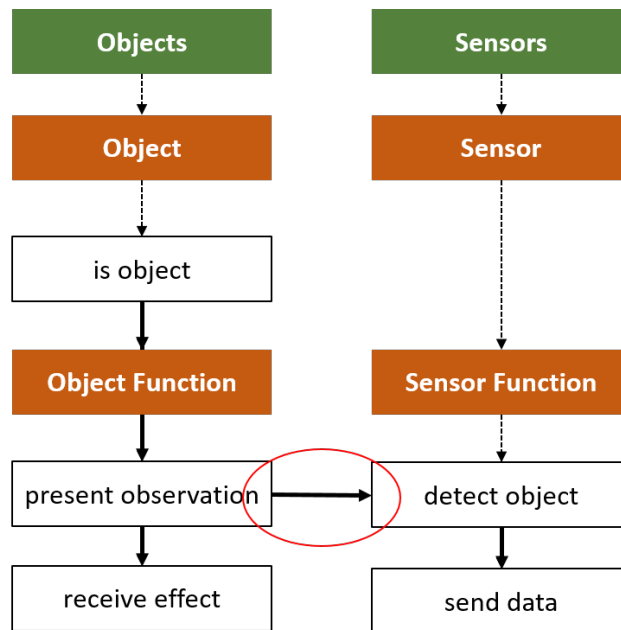


Figure 4.65. Object-to-Effect Model interaction of objects with sensors.

Table 4.12. Object-to-Effect Model interaction of objects with sensors depends upon whether the object presents some form of observation that is detectable by the sensors.

Ref	Abstraction-Level
	Object-To-Effect
3	Object presentation precedes the sensor detection of the object.

MP code for the coordination of the objects and sensors: This code includes conditional properties such that if the number of sensors is equal to the number of objects, each sensor operates on each object. Otherwise, each sensor operates on any unattended object, with the restriction that each sensor operates only on one object at any given instance. The code is listed as follows:

MP code segment 4.42: Coordination of the objects and sensors.

```
/* Interaction #3: Presentation precedes detection */
/* Each sensor has one sample of an object */
/**/
IF (#Object_Function == #Sensor_Function) THEN
  COORDINATE $p:(present_observation | not_present_observation),
    $d:(detect_object | not_detect_object)
    DO ADD $p PRECEDES $d; OD;
ELSE
  COORDINATE $p:(present_observation | not_present_observation)
  DO COORDINATE $d:(detect_object | not_detect_object)
    DO IF (#(present_observation | not_present_observation)
      THEN
        ADD $p PRECEDES $d; FI; OD; OD;
    BEFORE $d == 0 )
  FI;
```

4.3.2.3.3 Coordination of Sensors to Processors: As illustrated in Figure 4.66 and Table 4.13 this coordination includes only one interaction: the sensor data transmission precedes reception of that data by the processor.

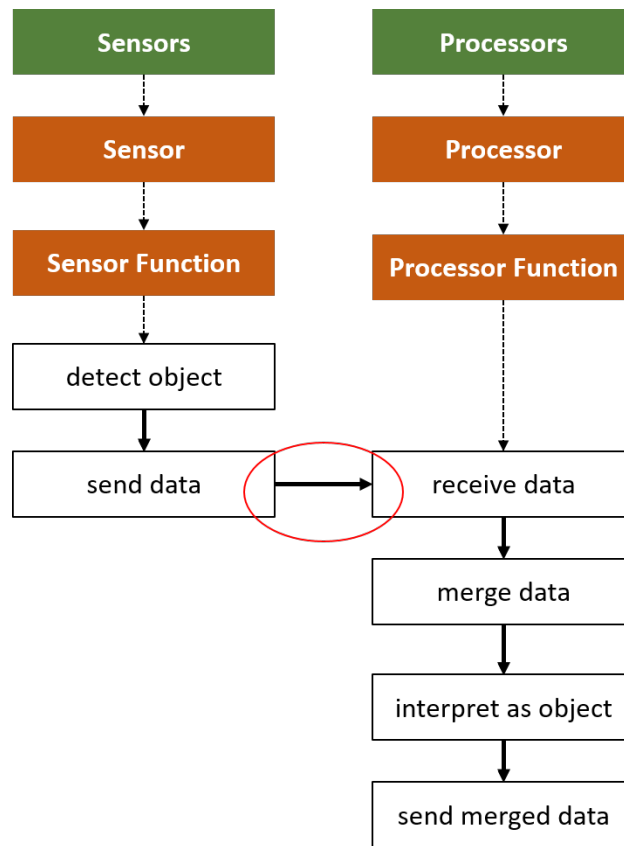


Figure 4.66. Object-to-Effect Model interaction of sensors with processors depends upon sending the data from the sensor and receiving the data at the processor.

Table 4.13. Object-to-Effect Model interaction of sensors with processors.

Ref	Abstraction-level
	Object-To-Effect
4	Sensor data transmission precedes reception.

MP code for the coordination of the sensor and processor: This coordination is also conditional, such that each processor function operates only on data from one object in any given instance. And so if the number of sensors is equal to the number of processors, each processor operates on data from only one of the sensors. Otherwise, a processor may have multiple sensors provide data from a given object. Also, not sending data can only be followed by not receiving data. The code is listed as follows:

MP code segment 4.43: Coordination of sensors and processors.

```

/* Interaction #4: Send Sensor data precedes its reception      */
/* Sensors may send multiple sets of data on the same object, */
/* but each processor operates on only one object              */
/**/
IF (#Sensor_Function == #Processor_Function) THEN
    COORDINATE      $Tx:(send_data      | not_send_data),
                    $Rx:(receive_data   | not_receive_data)
    DO ADD $Tx PRECEDES $Rx; OD;
ELSE
    COORDINATE      $f: Object_Function
    DO COORDINATE $p:(present_observation | not_present_observation)
        SUCH THAT $p IN $f
    DO COORDINATE $Tx: send_data
    DO COORDINATE $Rx:(receive_data | not_receive_data)
    DO IF ( ( #(send_data | not_send_data) BEFORE $Rx == 0 ) OR
            ( $p BEFORE $Tx AND $p BEFORE $Rx ) ) THEN
        ADD $Tx PRECEDES $Rx; FI; OD; OD; OD; OD;
    COORDINATE      $f: Object_Function
    DO COORDINATE $p:(present_observation | not_present_observation)
        SUCH THAT $p IN $f
    DO COORDINATE $nTx: not_send_data
    DO COORDINATE $nRx: not_receive_data
    DO IF ( ( #(send_data | not_send_data) BEFORE $nRx == 0 ) OR
            ( $p BEFORE $nTx AND $p BEFORE $nRx ) ) THEN
        ADD $nTx PRECEDES $nRx; FI; OD; OD; OD; OD;
    FI;

```

4.3.2.3.4 Coordination of Processors to Actors: As illustrated in Figure 4.67 and Table 4.14 this coordination includes only one interaction: The processor send of merged data to the actor precedes the actor reception of those data.

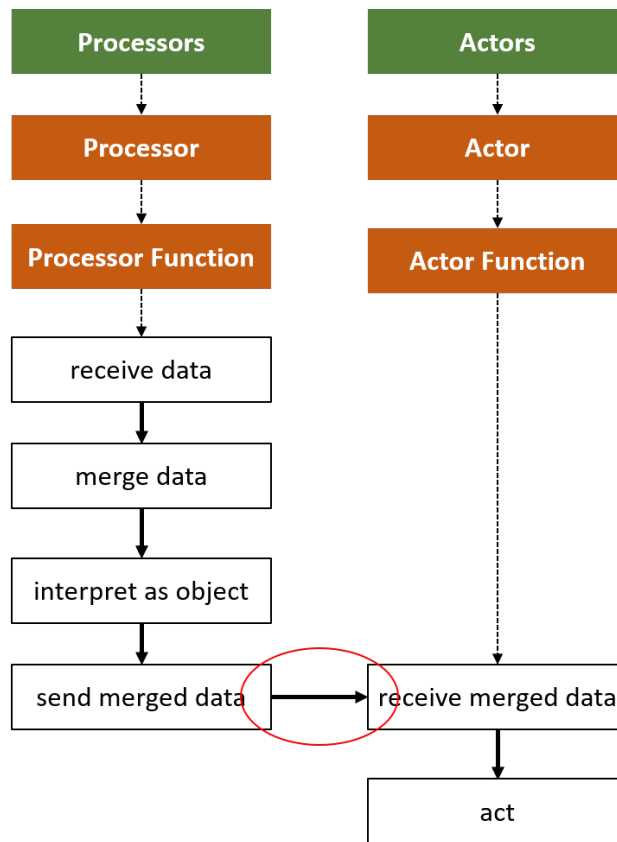


Figure 4.67. Object-to-Effect Model interaction of processors with actors depends upon sending the merged data from the processor and receiving those data by the actor.

Table 4.14. Object-to-Effect Model interaction of processors with actors.

Ref	Abstraction-Level
	Object-To-Effect
5	Merged data transmission precedes reception

MP code for the coordination of the processor and actor: This code is also conditional, such that each actor function operates only on data from one object in any given instance. And so if the number of processors is equal to the number of actors, each actor operates on data from only one of the processors. Otherwise, an actor may have multiple processors provide data from a given object. Further, not sending data can only be followed by not receiving data. The code is listed as follows:

MP code segment 4.44: Coordination of processors and actors.

```

/* Interaction #5: Send track precedes reception */
/**
IF (#Processor_Function == #Actor_Function) THEN
COORDINATE $Tx:(send_merged_data | not_send_merged_data ),
$Rx:(receive_merged_data | not_receive_merged_data)
DO ADD $Tx PRECEDES $Rx; OD;
ELSE
COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
SUCH THAT $p IN $f
DO COORDINATE $Tx:(send_merged_data)
DO COORDINATE $Rx:(receive_merged_data | not_receive_merged_data)
DO
IF ( (#(send_merged_data | not_send_merged_data )
BEFORE $Rx == 0 ) OR
($p BEFORE $Tx AND $p BEFORE $Rx) ) THEN
ADD $Tx PRECEDES $Rx; FI; OD; OD; OD; OD;
COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
SUCH THAT $p IN $f
DO COORDINATE $nTx: not_send_merged_data
DO COORDINATE $nRx: not_receive_merged_data
DO
IF ( (#(send_merged_data | not_send_merged_data )
BEFORE $nRx == 0 ) OR
($p BEFORE $nTx AND $p BEFORE $nRx) ) THEN
ADD $nTx PRECEDES $nRx; FI; OD; OD; OD; OD;
FI;

```

4.3.2.3.5 Coordination of Actors to Objects: As illustrated in Figure 4.68 and Table 4.15, this coordination includes only one interaction: The action upon the object precedes the effect on the object.

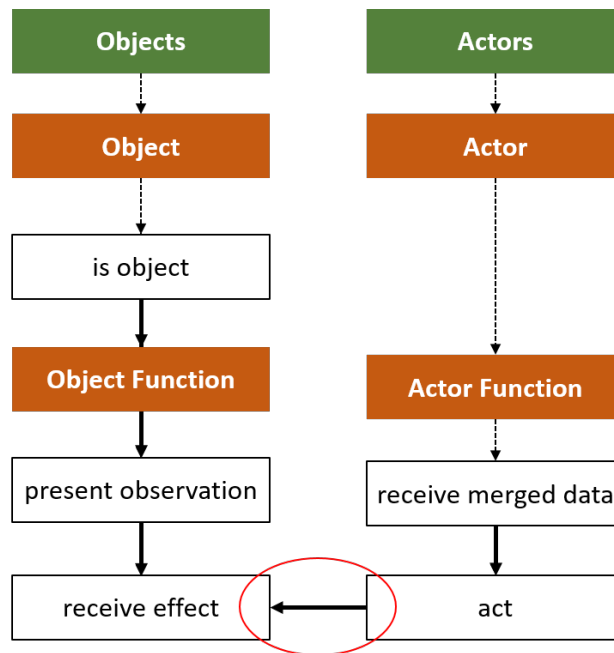


Figure 4.68. Object-to-Effect Model interaction of actors with objects depends upon the action having a direct effect upon the object.

Table 4.15. Object-to-Effect Model interaction of actors with objects.

Ref	Abstraction-Level
	Object-To-Effect
6	Action precedes the effect on the object.

MP code for the coordination of the actor and object: This code captures the behavior such that not receiving an effect must follow a non-action, and multiple actions may be imposed on a single object. The code is listed as follows:

MP code segment 4.45: Coordination of actors and objects.

```

/* Interaction #6: Action precedes effect */
/**
COORDINATE    $f: Object_Function
DO COORDINATE $p:(present_observation | not_present_observation)
    SUCH THAT $p IN $f
    DO COORDINATE $s: act
    DO COORDINATE $h:(receive_effect | not_receive_effect)
    DO IF ($p BEFORE $s AND $p BEFORE $h) THEN
        ADD $s PRECEDES $h; FI; OD; OD; OD; OD;

COORDINATE    $f: Object_Function
DO COORDINATE $p:(present_observation | not_present_observation)
    SUCH THAT $p IN $f
    DO COORDINATE $s: not_act
    DO COORDINATE $h: not_receive_effect
    DO IF ($p BEFORE $s AND $p BEFORE $h) THEN
        ADD $s PRECEDES $h; FI; OD; OD; OD; OD;

```

4.3.2.3.6 A Functional Flow Block Diagram (FFBD) of the Object-to-Effect Model: With the definition of event alternatives and interactions among the systems, the FFBD can be constructed, as shown in Figure 4.69. This figure depicts each of the alternative events, the functions, and hierarchy of the model structure.

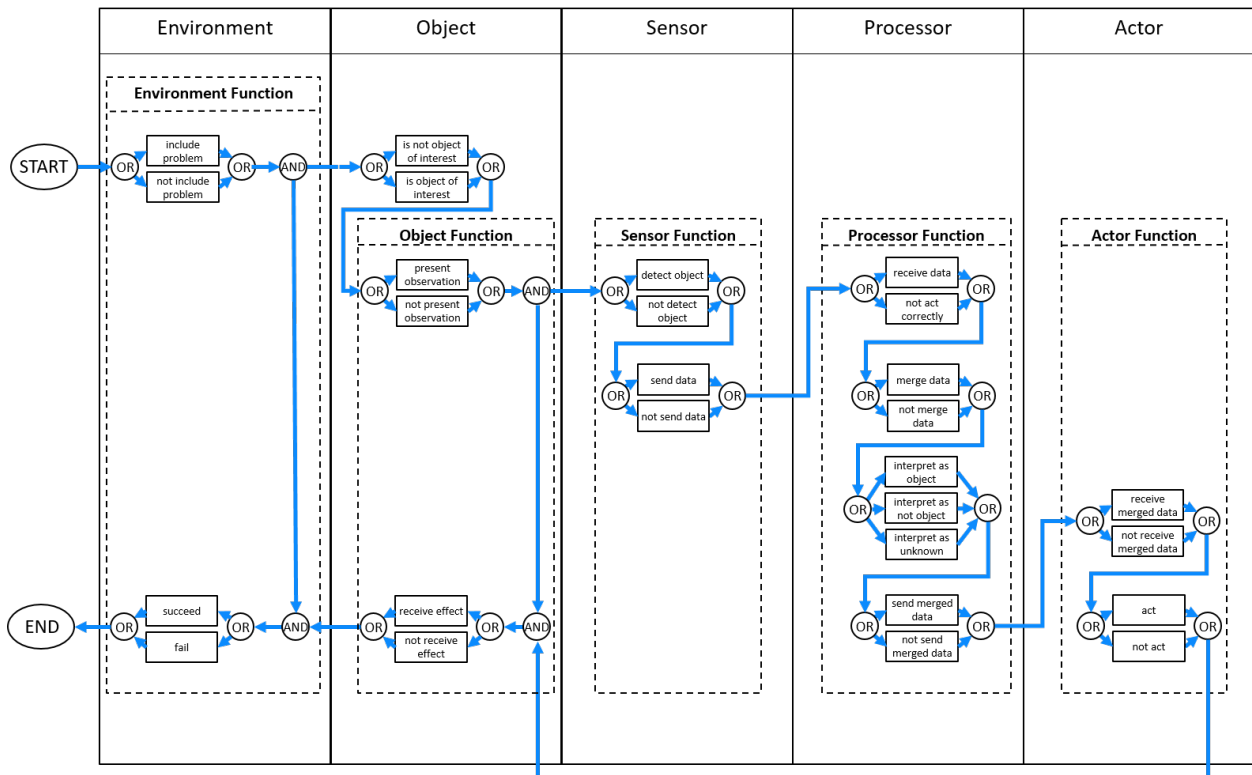


Figure 4.69. FFBD of the Object-to-Effect Model.

4.3.2.4 Object-to-Effect Model: Constraint derivation

Sixteen constraints of the model were derived by interactively executing the model, and considering the results. These constraints included logical, definition, and design types. For one object, one sensor, one processor, and one actor, denoted as (1-1-1-1), the unconstrained model produced 12,288 possible traces. After the constraints were applied, the number of valid traces was reduced to eighteen valid traces. The following lists the constraints:

1. *Type: Definition* - Definition of the problem state.
2. *Type: Definition* - Definition of success and failure.
3. *Type: Logical* - If there exists no observable presentation, then there is no detection.
4. *Type: Logical* - If there is no detection, then the sensor sends no data.
5. *Type: Logical* - If there is no data sent by the sensor, then no data is received by the processor.
6. *Type: Logical* - If no data is received by the processor, then data cannot be merged.
7. *Type: Logical* - If there is no merged data, then no interpretation is possible by the processor.
8. *Type: Logical* - If there is no merged data, then the processor cannot send merged data.
9. *Type: Design* - If the processor received data on the object that is not of interest, do not interpret as the object of interest.
10. *Type: Design* - If the processor received data on the object that is of interest, do not interpret as an object not of interest.
11. *Type: Logical* - If the processor does not send merged data, then the actor does not receive merged data.
12. *Type: Logical* - If the actor takes no action, then the object is not acted upon.
13. *Type: Design* - If the merged data is not received, then do not take an action.
14. *Type: Design* - If the object is interpreted as not an object of interest or unknown, then do not act.
15. *Type: Design* - Each action has no more than one possible effect on an object.
16. *Type: Logical* - Without the presentation of an observation, the object cannot receive an effect from the actor.

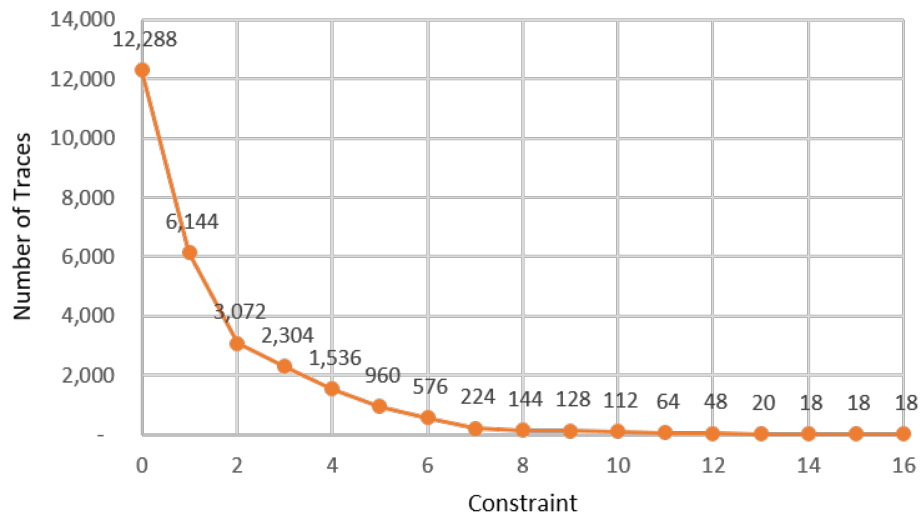


Figure 4.70. Constraints applied to the Object-to-Effect Model, for scope of one object, one sensor, one processor, and one actor, labelled as (1-1-1-1). The unconstrained model has 12,288 traces that are limited to just eighteen valid traces after applying constraints.

4.3.2.4.1 Constraints 1 and 2: These constraints define the problem state and outcome, as listed in Table 4.16. These constraints are applied to the interactions of the environment and the nature of the object and effects applied to the object.

Table 4.16. Object-to-Effect Model constraints 1 and 2: Definitions of the problem state and success.

Ref	Type of Constraint	Abstraction	Number of traces after cumulative constraints applied
		Object-To-Effect	
0	none	n/a	12,288
1	Definition	Definition of the problem state.	6,144
2	Definition	Definition of success and failure.	3,072

MP code segment 4.46: Constraints 1 and 2.

```

/*Constraint #1. DEFINITION: Problem state */
/**/
ENSURE (    #include_problem == 1 -> #is_of_interest >= 1);
ENSURE (#not_include_problem == 1 -> #is_of_interest == 0);

/*Constraint #2. DEFINITION: Success and failure */
/**/
IF (#failed_outcome == 1) THEN
    ENSURE EXISTS $x: Object_Function
        ((#is_of_interest BEFORE $x -
          #not_receive_effect IN $x == 0)
         OR
          (#is_not_of_interest BEFORE $x -
          #receive_effect IN $x == 0));
FI;

IF (#successful_outcome == 1) THEN
    ENSURE FOREACH $x:Object_Function
        (#is_of_interest BEFORE $x -
         #receive_effect IN $x == 0);

    ENSURE FOREACH $x:Object_Function
        (#is_not_of_interest BEFORE $x -
         #not_receive_effect IN $x == 0);
FI;

```

4.3.2.4.2 Constraints 3 and 4: As listed in Table 4.17, the constraints that relate the object and the sensor include that the sensor detection depends upon an observable presentation by the object and that sending any subsequent data depends upon that detection. These are both logical constraints.

Table 4.17. Object-to-Effect Model constraints 3 and 4: Logical exchange between the object and sensor.

Ref	Type of Constraint	Abstraction	Number of traces after cumulative constraints applied
		Object-To-Effect	
3	Logical	If no observable presentation, then there is no detection.	2,304
4	Logical	If there is no detection, then no data is sent.	1,536

MP code segment 4.47: Constraints 3 and 4.

```

/*Constraint #3. LOGICAL: If no observable presentation, */
/*                        then there is no detection */
/**/ ENSURE FOREACH $x: detect_object
    (#present_observation BEFORE $x > 0);

/*Constraint #4. LOGICAL: If there is no detection, then no */
/*                        data sent */
/**/ ENSURE FOREACH $x: send_data (#detect_object BEFORE $x > 0);

```

4.3.2.4.3 Constraints 5 through 10: These constraints relate data received by the processor and sent to the actor, as indicated in Table 4.18. The logical constraints rely upon receiving data from the sensors, merging the data from any of the sensor(s), and sending data to the actor(s). The design constraints include correct interpretation, given data on an object of interest or an object not of interest. These design requirements reflect 100% accuracy of this interpretation, and so the developer may decide to relax this constraint, in favor of a performance requirement such as “correct interpretation x% of the time.” Of course, this change to the schema will increase the number of valid traces, and the developer would then need to address the associated templates with inaccurate interpretation of the available data. The value associated with the probability of correct interpretation would then need to be reflected appropriately in the calculation of the probability of occurrence of each trace, per the description in section 4.3.2.6.

Table 4.18. Object-to-Effect Model constraints 5 through 10: Limitations applied to the processor functions.

Ref	Type of Constraint	Abstraction	Number of traces after cumulative constraints applied
		Object-To-Effect	
5	Logical	If no data sent, then no data received.	960
6	Logical	If no data is received, then cannot merge data	576
7	Logical	If there is no data merged, then no interpretation possible.	224
8	Logical	If there is no data merged, then cannot send the merged data.	144
9	Design	If received data on the non-object of interest, do not interpret as the object.	128
10	Design	If received data on the object of interest, do not interpret as a non-object.	112

MP code segment 4.48: Constraints 5 through 10.

```

/*Constraint #5. LOGICAL: If no data sent, then no data received*/
/**/ ENSURE FOREACH $x: receive_data (#send_data BEFORE $x > 0);

/*Constraint #6. LOGICAL: If no data is received, then cannot */
/* merge data */
/**/ ENSURE FOREACH $x: merge_data (#receive_data BEFORE $x > 0);

/*Constraint #7. LOGICAL: If there is no merged data, then */
/* is no interpretation */
/**/ ENSURE FOREACH $x: not_merge_data
    (#interpret_as_unknown AFTER $x == 1);
/**/ ENSURE FOREACH $x: merge_data
    (#interpret_as_unknown AFTER $x == 0);

/*Constraint #8. LOGICAL: If no merged data, then cannot */
/* send merged data */
/**/ ENSURE FOREACH $x: not_merge_data
    (#send_merged_data AFTER $x == 0);

/*Constraint #9. DESIGN REQUIREMENT: If received data on a */
/* friendly, then don't interpret as hostile */
/**/ ENSURE FOREACH $x: is_not_of_interest, $y: receive_data
    ($x BEFORE $y -> #interpret_as_object AFTER $x == 0);

/*Constraint #10. DESIGN REQUIREMENT: If received data on a */
/* hostile, then don't interpret as friendly */
/**/ ENSURE FOREACH $x: is_of_interest, $y: receive_data
    ($x BEFORE $y -> #interpret_as_not_object AFTER $x == 0);

```

4.3.2.4.4 Constraints 11 through 16: These constraints relate to the actor and object, as outlined in Table 4.19. Constraints 15 and 16 have an effect for multiple objects, and so for scope of (1-1-1-1), there is no further reduction of the number of valid traces.

Table 4.19. Object-to-Effect Model constraints 11 through 16: Limitations on the actions applied to the object.

Ref	Type of Constraint	Abstraction	Number of traces after cumulative constraints applied
		Object-To-Effect	
11	Logical	If the merged data is not sent, then cannot receive merged data.	64
12	Logical	If there is no action, then the object is not acted upon.	48
13	Design	If the merged data is not received, then do not act.	20
14	Design	If interpreted as not the object of interest or unknown, then do not act.	18
15	Design	Each action has no more than one possible effect	18
16	Logical	Without the presentation of an observation, cannot receive an effect	18

MP code segment 4.49: Constraints 11 through 16.

```
/*Constraint #11. LOGICAL: If the merged data is not sent, then*/  
/*merged data not received*/  
/**/ ENSURE FOREACH $x:receive_merged_data  
      (#send_merged_data BEFORE $x > 0);  
  
/*Constraint #12. LOGICAL: If there is no action, then the*/  
/*object does not receive an effect*/  
/**/ ENSURE FOREACH $x:receive_effect      (#act BEFORE $x > 0);  
  
/*Constraint #13. DESIGN REQUIREMENT: If no data received, then*/  
/*do not act*/  
/**/ ENSURE FOREACH $x:act      (#receive_merged_data BEFORE $x > 0);  
  
/*Constraint #14. DESIGN REQUIREMENT: If interpreted as a*/  
/*friendly, or unknown, then don't act*/  
/**/ ENSURE FOREACH $x:interpret_as_not_object(#act AFTER $x==0);  
/**/ ENSURE FOREACH $x:interpret_as_unknown      (#act AFTER $x==0);  
  
/*Constraint #15. DESIGN REQUIREMENT: Each action has no more*/  
/*than one effect*/  
/**/ ENSURE FOREACH $x:act      (#receive_effect AFTER $x <= 1);  
  
/*Constraint #16. LOGICAL: If not present observation then not*/  
/*receive an effect*/  
/**/ ENSURE FOREACH $x:Object_Function  
      (#receive_effect IN $x +  
       #not_present_observation IN $x <= 1);
```

4.3.2.5 Object-to-Effect Model: Pattern identification

Pattern identification is accomplished by aligning each of the events in the model with attributes that follow a semantic definition, as aligned to a repository.

4.3.2.5.1 Semantic attributes: These attributes included Boolean expressions for each of the events. Additionally, statistics are employed to track whether each event is in a template, and the number of non-compliant events (i.e., events not in a template) within each trace and within the entire schema. The next sections discuss how each of the attributes are applied in the model. As indicated in the following MP code, most of the attributes identify the Boolean expression of the attribute with the exception of the count of events.

MP code segment 4.50: Attributes associated with each event.

```

/* Attribute definitions indicated as Boolean expressions      */
/* These attributes serve as meta-data for the template query */
ATTRIBUTES{bool isEnvironmentFunction,
              isObjectFunction,
              isSensorFunction,
              isProcessorFunction,
              isActorFunction,
              favorable, unfavorable,
              Ex, nEx, /*existence of problem*/
              It, nIt, /*object of interest */
              Pr, nPr, /*present observation */
              Se, Ob, Or, De, Ac, /*OODA */
              nSe, nOb, nOr, nDe, nAc, /*broken OODA */
              Tx, Rx, nTx, nRx, /*transmit, receive, */
              Re, Int, nRe, nInt, /*relay, interrupt */
              Eff, nEff, /*effect on Object */
              fl, scs; /*failed or successful*/

};

/* Attributes used to identify whether events are in a template*/
ATTRIBUTES{bool inTemplate; /*indication of event in template */
           number count, /*number of events not in template*/
           tracecount; /*number of traces with events not*/
           /*in any template*/

};

```

Attributes aligned to functions: Table 4.20 characterizes the attributes for each of the functions of the model. This construct enables a set of events to be included within the respective function, such that the set of events can be queried by the model as a unit, keeping the association as a single function. This table also lists the favorable/unfavorable attributes and whether the event is in a template.

Table 4.20. Object-to-Effect Model functional attributes enable a set of events to be aligned to each function. Also listed are the favorable/unfavorable attributes, and the indication of whether an event is in a template.

Attribute	Description	Attributes Applied to Events
isEnvironmentFunction	Events associated with the environment function	Environment_Function
isObjectFunction	Events associated with the object function	Object_Function
isSensorFunction	Events associated with the sensor function	Sensor_Function
isProcessorFunction	Events associated with the processor function	Processor_Function
isActorFunction	Events associated with the actor function	Actor_Function
favorable, unfavorable	Identification of positive or negative events	all events within each function
inTemplate	identification of whether event is in a template	all events within each function

Attributes associated with the environment: These attributes include whether a problem exists (**Ex**, **nEx**) and whether the effect was successful (**scs**, **fl**) for all of the objects, as indicated in Table 4.21 and Figure 4.71.

Table 4.21. Object-to-Effect Model environmental attributes, such that the object exists or not exist, and the outcome fails or succeeds.

Attribute	Description	Application of the Attribute to Events
Ex , nEx	Existence, or non-existence of a problem	include_problem, not_include_problem
fail , success	failed or successful outcome	failed_outcome, successful_outcome

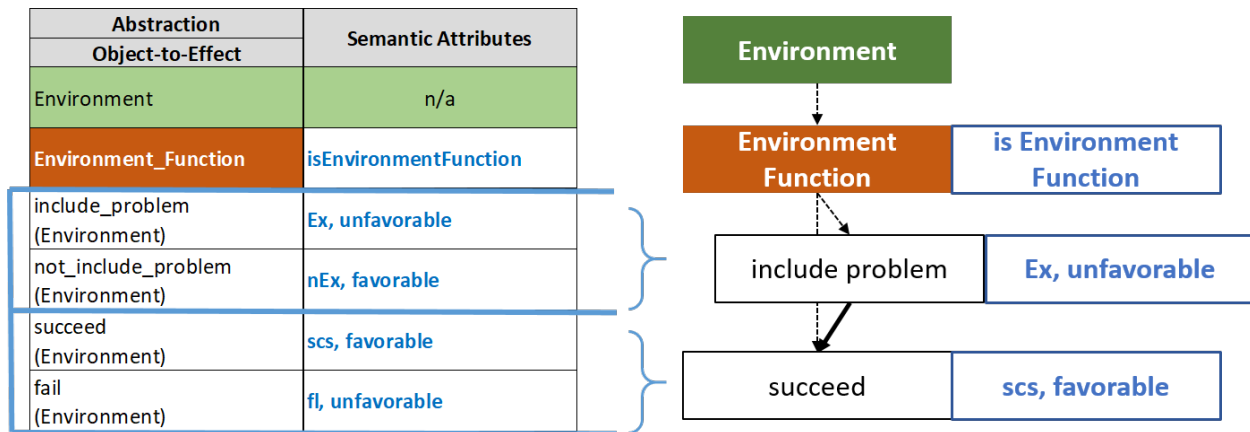


Figure 4.71. Object-to-Effect Model environmental attributes aligned to each event in the environment function, with an example illustrated in the associated figure to the right of the table.

MP code segment 4.51: Environment attribute assignments.

```

not_include_problem :
    BUILD{nEx:= true;    favorable:=true; inTemplate:=true;};
include_problem :
    BUILD{ Ex:= true; unfavorable:=true; inTemplate:=true;};

successful_outcome :
    BUILD{success:= true;    favorable:=true; inTemplate:=true;};
failed_outcome :
    BUILD{fail    := true; unfavorable:= true;inTemplate:=true;};

```

Attributes associated with the objects: These attributes include whether the object is of interest (**It**, **nIt**) and whether the effect on each individual object of interest (**Eff**, **nEff**) occurred, as indicated

in Table 4.22 and Figure 4.72.

Table 4.22. Object-to-Effect Model object attributes, implemented such that the object exists or does not exist, and the object functions include whether to present an observation and whether the effect occurs.

Attribute	Description	Application of the Attribute to Events
It, nIt	Identity of the object, whether the object is of interest	is_of_interest, is_not_of_interest
Pr, nPr	Indication of whether the object is potentially exposed by a sensor	present_observation, not_present_observation
Eff, nEff	Effect upon the object	receive_effect, not_receive_effect

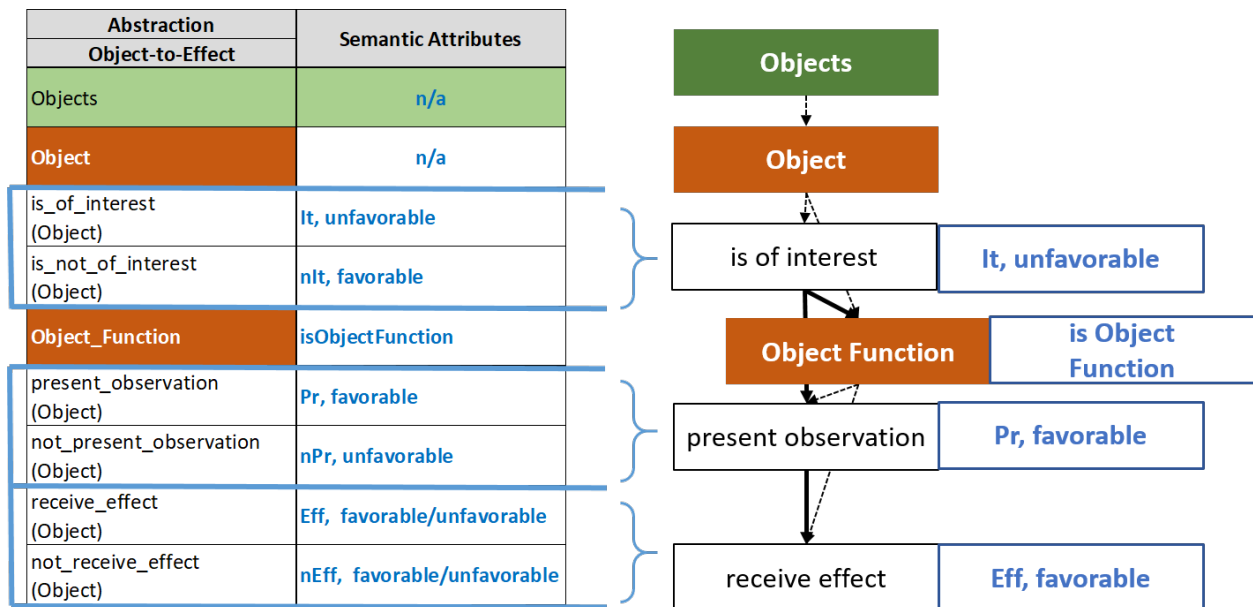


Figure 4.72. Object-to-Effect Model object attributes aligned to each of the associated events, with a particular instance depicted to the right of the table.

MP code segment 4.52: Object attribute assignments.

```

is_not_of_interest :
    BUILD{ nIt:= true;   favorable:= true;};
is_of_interest :
    BUILD{ It:= true; unfavorable:= true;};

present_observation :
    BUILD{ Pr:= true;   favorable:= true;};
not_present_observation :
    BUILD{ nPr:= true; unfavorable:= true;};

receive_effect :
    BUILD{ Eff:= true;};
not_receive_effect :
    BUILD{ nEff:= true;};

```

Attributes associated with communication and the OODAloop: These attributes include whether data transmitted (**Tx**, **nTx**) and received (**Rx**, **nRx**), as indicated in Table 4.23. The OODA loop is depicted in Table 4.24 and the chain of events to include observe (**Ob**, **nOb**) or sense (**Se**, **nSe**), orient (**Or**, **nOr**), decide (**De**, **nDe**), and act (**Ac**, **nAc**).

Figures 4.73 and 4.74, followed by the MP code listing outline the derivation and application of these attributes within the model. Note that the action may be favorable or unfavorable depending upon whether the object is of interest and the subsequent interpretation of the data by the processor, and so conditional attributes were applied to the model for the action.

Table 4.23. Object-to-Effect Model communication attributes consist of Boolean representation of transmit and receive. Notice that the identical attributes were applied to sending/receiving data and sending/receiving merged data.

Attribute	Description	Application of the Attribute to Events
Tx , nTx	Transmit data	send_data, send_merged_data, not_send_data, not_send_merged_data
Rx , nRx	Receive data	receive_data, receive_merged_data, not_receive_data, not_receive_merged_data

Table 4.24. Object-to-Effect Model OODA loop attributes include sensing or observing, orienting, deciding, and acting. The Object-to-Effect model distributes these events across the sensor, processor, and actor functions of the model.

Attribute	Description	Application of the Attribute to Events
Se, nSe	Sense or not sense: indication of whether perceive the object of interest	detect_object
Ob, nOb	Observe or not observe: synonymous with sense or not sense	detect_object
Or, nOr	Orient or not orient: based on perceptions	merge_data
De, nDe	Decide or not decide: based on processing available data	interpret_as_object, interpret_as_non-object, interpret_as_unknown
Ac, nAc	Act or not act: based on the decision	act, non_act

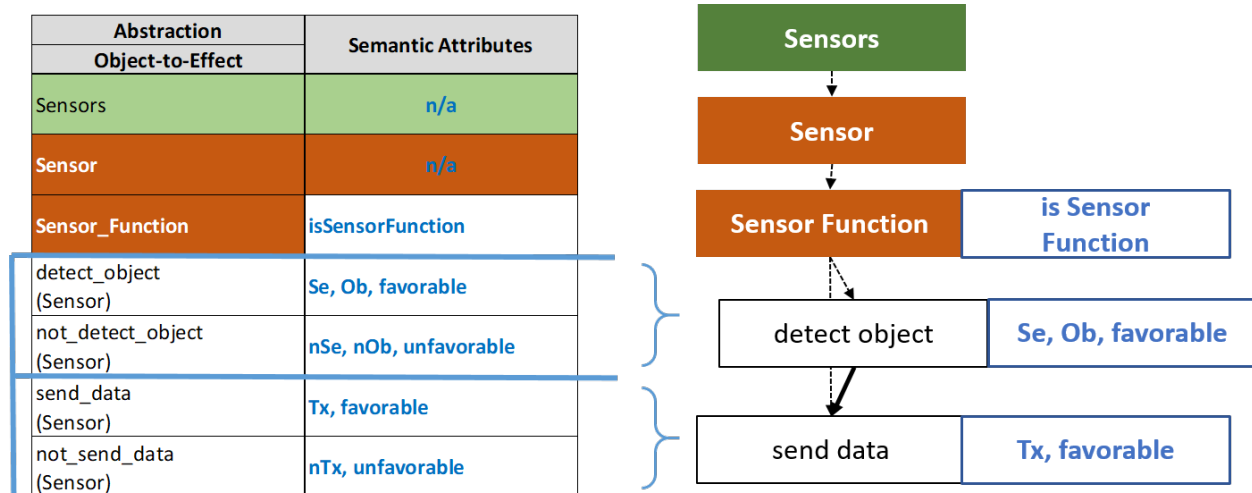


Figure 4.73. Object-to-Effect Model sensor attributes include sense or observe and sending the data to the processor. Note that in this case, the developer aligned both the sense (Se, nSe) and observe (Ob, nOb) attributes to the detection event.

MP code segment 4.53: Sensor attributes.

```

detect_object :
  BUILD{ Ob:= true; Se:= true;   favorable:= true;};
not_detect_object :
  BUILD{ nOb:= true;nSe:= true; unfavorable:= true;};

send_data :
  BUILD{ Tx:= true;   favorable:= true;};
not_send_data :
  BUILD{ nTx:= true; unfavorable:= true;};

```

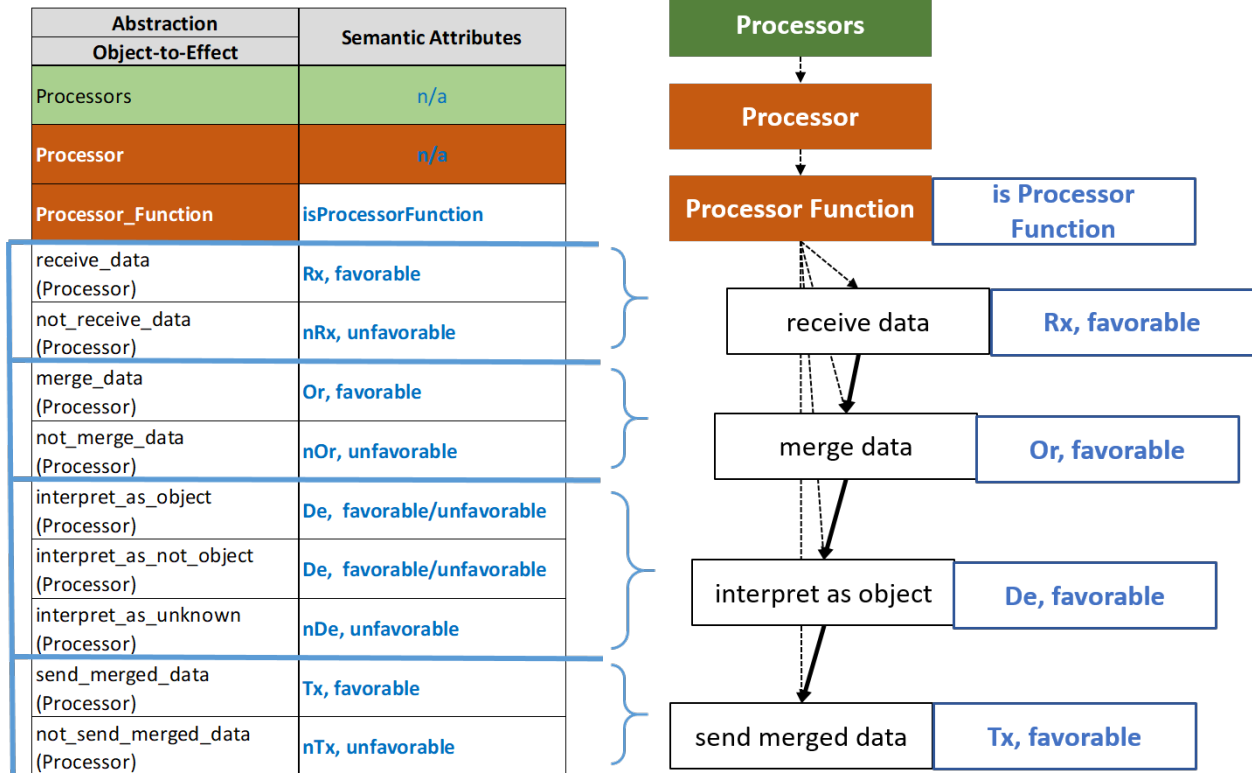


Figure 4.74. Object-to-Effect Model processor attributes include receiving data, orienting to the data, deciding the interpretation, and sending the merged data.

MP code segment 4.54: Processor attributes.

```

receive_data :
    BUILD{ Rx:= true;    favorable:= true;};
not_receive_data :
    BUILD{ nRx:= true; unfavorable:= true;};

merge_data :
    BUILD{ Or:= true;    favorable:= true;};
not_merge_data :
    BUILD{ nOr:= true; unfavorable:= true;};

interpret_as_not_object :
    BUILD{ De:= true; };
interpret_as_object :
    BUILD{ De:= true; };
interpret_as_unknown :
    BUILD{ nDe:= true; unfavorable:= true;};

send_merged_data :
    BUILD{ Tx:= true;    favorable:= true;};
not_send_merged_data :
    BUILD{ nTx:= true; unfavorable:= true;};

```

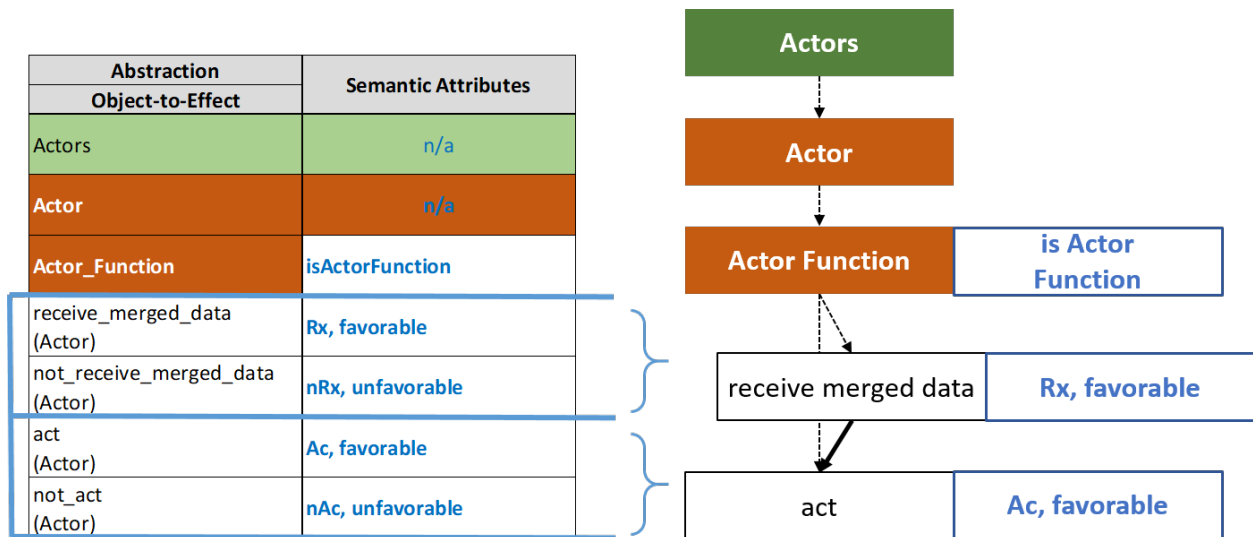


Figure 4.75. Object-to-Effect Model actor attributes aligned to receiving the merged data taking the action.

MP code segment 4.55: Actor attributes.

```

receive_merged_data :
    BUILD{ Rx:= true;    favorable:= true;};
not_receive_merged_data :
    BUILD{nRx:= true; unfavorable:= true;};

act :
    BUILD{ Ac:= true; };
not_act :
    BUILD{nAc:= true; };

```

MP code segment 4.56: Conditional favorable and unfavorable attributes associated with acting or not acting.

```

COORDINATE $x: (is_of_interest | is_not_of_interest)
DO COORDINATE $y: (act | not_act)
DO
    IF ($x IS is_of_interest AND $y IS act AND $x BEFORE $y)
    THEN $y.favorable := true; FI;

    IF ($x IS is_of_interest AND $y IS not_act AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;

    IF ($x IS is_not_of_interest AND $y IS act AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;

    IF ($x IS is_not_of_interest AND $y IS not_act
        AND $x BEFORE $y)
    THEN $y.favorable := true; FI;
OD; OD;

COORDINATE $x: ( interpret_as_unknown
                | interpret_as_object
                | interpret_as_not_object)
DO COORDINATE $y: (act | not_act)
DO
    IF ($x IS interpret_as_unknown AND $y IS not_act
        AND $x BEFORE $y)
    THEN $y.favorable := true; FI;

    IF ($x IS interpret_as_unknown AND $y IS act
        AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;
OD; OD;

```

4.3.2.5.2 The topology: Populated the model with associated attributes enable the developer to derive templates for patterns of behavior of the model. Figure 4.76 outlines the sequence of events leading to an effect upon an object of interest. These attributes function as meta-data, such that a set of model applications (e.g., air-space protection, rescue, traffic control, or home security models) have the same underlying structure as defined by the topology and by aligning the domain-specific events to consistent abstract-level attributes. Notice from the figure that the templates operate on the shaded sequence of events, while the environment establishes the problem and interprets the results.

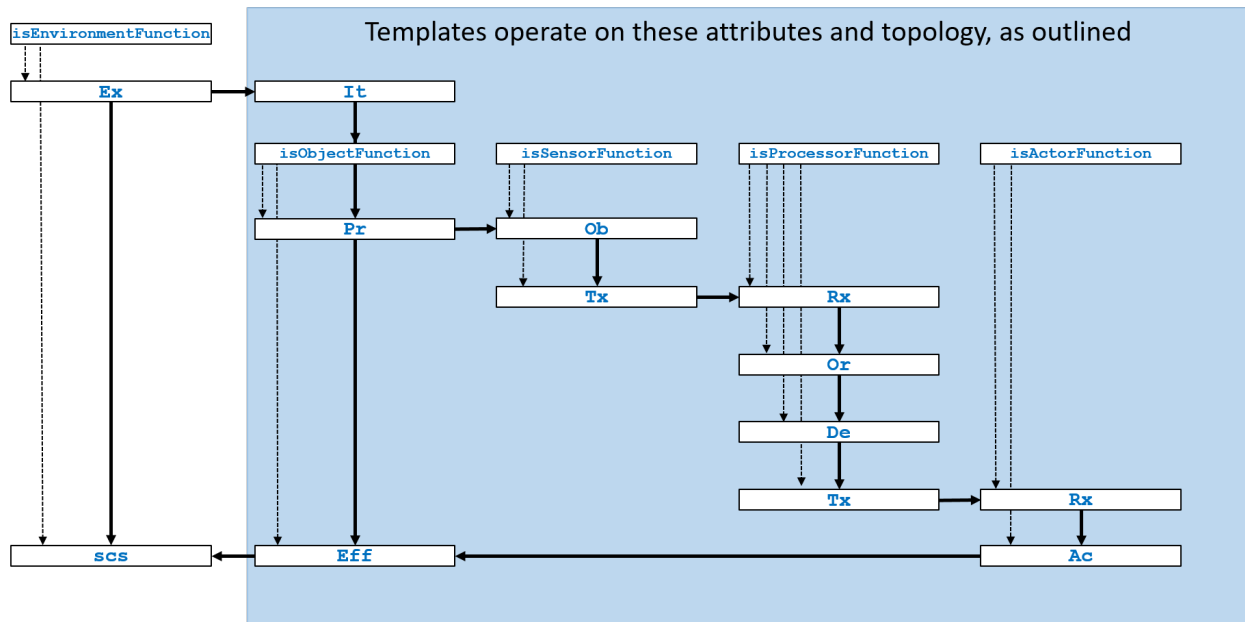


Figure 4.76. Object-to-Effect model topology with attributes assigned to each event.

4.3.2.5.3 Templates: These templates were developed by executing the model at the initial scope of (1-1-1-1) with one object, one sensor, one processor, and one actor, then systematically increasing the scope to include multiple sets of these objects and their associated functions. It became critical to operate on the sets of events as functions, so that the pattern identified only the events within that particular function. Table 4.25 lists all of the templates derived for the model. The first eighteen were identified at scope of (1-1-1-1), and the next four were derived by systematically increasing the scope. The following code, for Template 9, illustrates the implementation of the template by using the attributes labeled as “isObjectFunction”, “isSensorFunction”, “isProcessorFunction”, and “isActorFunction”, respectively. Each of the events queried by the template are included within these functions.

Figure 4.77 illustrates the trace that satisfies Template 9. Notice that this figure shows the indication that all events in all traces were included in a template. This indication verifies that for the scope of execution, the developer has confidence that the results have been evaluated and produce the expected outcomes. For the scope of (1-1-1-1), with eighteen traces, the developer can easily review the outcomes; however, at a higher scope of execution, this method provides an exhaustive and computational means to query and evaluate the results.

Table 4.25. Object-to-Effect Model templates 1 through 18 were identified at scope of (1-1-1-1), indicating one object, one sensor, one processor, and one actor. Template 19 was derived at a scope of (2-1-1-1), with an increased object. Template 20 was derived at a scope of (1-2-1-1), with an increased sensor only. Template 21 was derived at a scope of (1-1-2-1), with an increased processor only. Template 22 was derived at a scope of (1-1-1-2), with an increased actor only.

Type	Ref.	Application
		Object-to-Effect
Model-specific Templates	1	Identified as not an object of interest
	2	Non-object, merged data not received
	3	Non-object, merged data not sent
	4	Non-object, interpreted as unknown
	5	Non-object, data not received
	6	Non-object, data not sent
	7	Non-object, not detected
	8	Non-object, not observable
	9	Object receives effect
Model-specific Templates	10	Object acted upon without effect
	11	Object not acted upon
	12	Object, merged data not received
	13	Object, merged data not sent
	14	Object interpreted as unknown
	15	Object, data not received
	16	Object, data not sent
	17	Object not detected
	18	Object not observable
Model-specific Warnings	19	Isolated object
	20	Isolated sensor
	21	Isolated processor
	22	Isolated shooter

MP code segment 4.57: Template 9: Object receives effect.

```

/* Template 9: Object of interest receives effect *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.Rx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.Ac
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.Eff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 9:"); MARK;
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

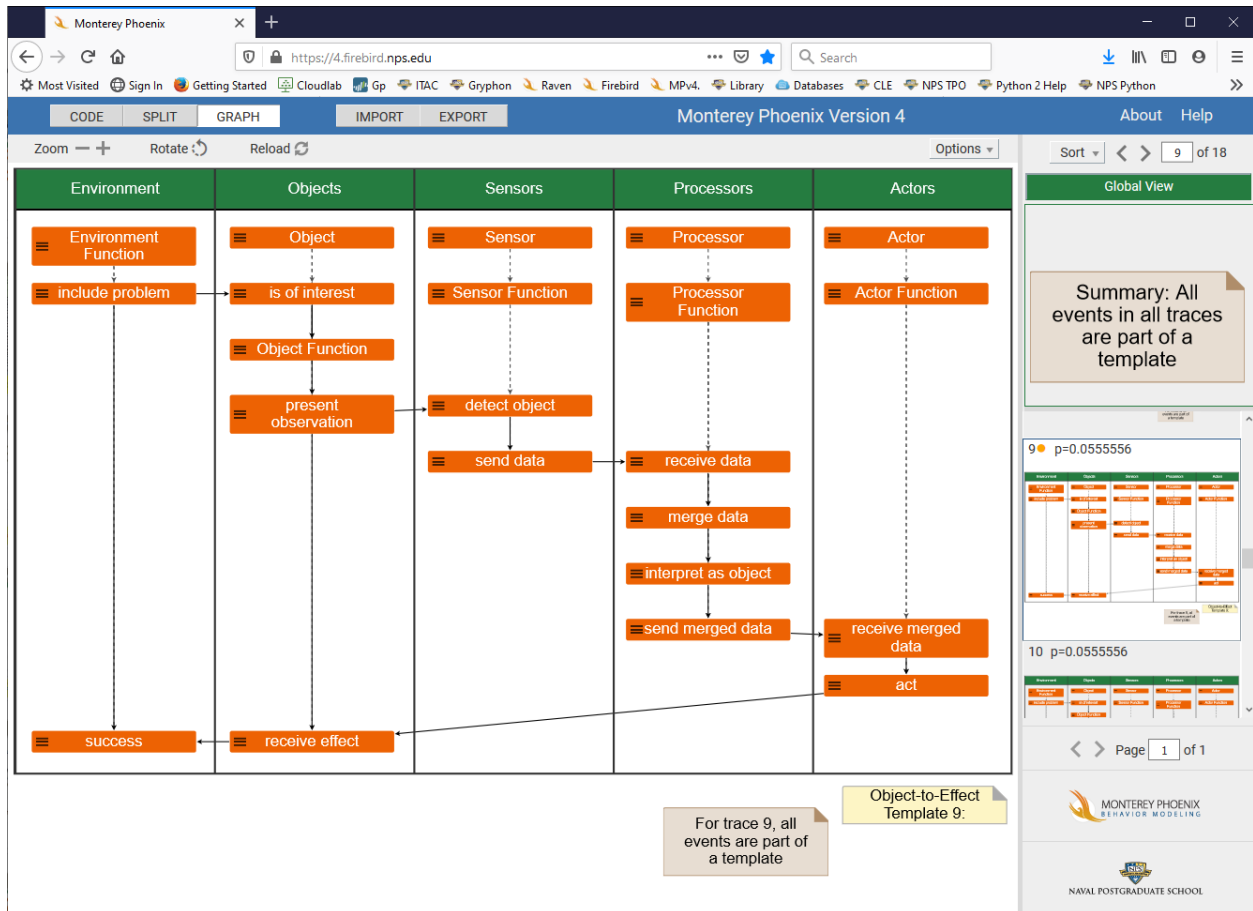


Figure 4.77. Object-to-Effect Model Template 9 output executed at a scope of (1-1-1-1), with one object, one sensor, one processor, and one actor. Notice the global view summary, indicating that all events in all traces are part of a template. This provides an exhaustive means to verify that the model produces the expected results.

4.3.2.6 Object-to-Effect Model: analysis

4.3.2.6.1 Run matrix: Appendix A.2 illustrates the output associated with Templates 1-22. Also indicated in Table 4.26, Templates 1-18 were derived from runs 1-18, at scope (1-1-1-1). For run 2, adding an additional object at scope (2-1-1-1), resulted in a new discovery since only one set of sensors, processors, and actors were available to address two objects. This condition was captured by Template 19, as an isolated object that was not addressed by a sensor. Similarly, runs 3-5 add an additional sensor, processor, and actor, labeled as (1-2-1-1), (1-1-2-1), and (1-1-1-2), respectively. These runs derive Templates 20-22.

All remaining runs, 6 through 20, consist of combinations of Templates 1-22, with all events from all traces being part of one or more of these templates. Considering all outcomes of the run matrix, 77,377,536 potential outcomes were systematically reduced to 3,491 valid traces that complied with the logical, design, and definition constraints. Since all events of all traces were combinations of the 22 templates, the developer has confidence that there are no combinations of events left unaccounted. Therefore, all possible behaviors have been exhaustively analyzed, up to the scope of execution of each run in the matrix.

Table 4.26. Object-to-Effect run matrix.

#	Run Notation	Number of Each				Number of Combinations of Alternatives for Each Set of Functions					Number of Potential Traces	Number of Valid Traces		
		Environment	Objects	Sensors	Processors	Actors	Environment	Objects	Sensors	Processors				Actors
1	(1-1-1-1)	1	1	1	1	1	2*2 = 4	2*(2*2) = 8	2*2 = 4	2*2*2*3 = 24	2*2 = 4	12,288	18	Templates 1-18
2	(2-1-1-1)	1	2	1	1	1	4	64	4	24	4	98,304	72	Combinations of Templates 1-18 and New Templates 19-22
3	(1-2-1-1)	1	1	2	1	1	4	8	16	24	4	49,152	70	
4	(1-1-2-1)	1	1	1	2	1	4	8	4	576	4	294,912	66	
5	(1-1-1-2)	1	1	1	1	2	4	8	4	24	16	49,152	30	
6	(2-2-1-1)	1	2	2	1	1	4	64	16	24	4	393,216	204	Combinations of Templates 1-22
7	(1-2-2-1)	1	1	2	2	1	4	8	16	576	4	1,179,648	118	
8	(1-1-2-2)	1	1	1	2	2	4	8	4	576	16	1,179,648	78	
9	(2-1-1-2)	1	2	1	1	2	4	64	4	24	16	393,216	120	
10	(1-2-1-2)	1	1	2	1	2	4	8	16	24	16	196,608	130	
11	(2-1-2-1)	1	2	1	2	1	4	64	4	576	4	2,359,296	264	
12	(1-2-2-2)	1	1	2	2	2	4	8	16	576	16	4,718,592	130	
13	(2-1-2-2)	1	2	1	2	2	4	64	4	576	16	9,437,184	312	
14	(2-2-1-2)	1	2	2	1	2	4	64	16	24	16	1,572,864	372	
15	(2-2-2-1)	1	2	2	2	1	4	64	16	576	4	9,437,184	300	
16	(2-2-2-2)	1	2	2	2	2	4	64	16	576	16	37,748,736	324	
17	(3-1-1-1)	1	3	1	1	1	4	512	4	24	4	786,432	288	
18	(1-3-1-1)	1	1	3	1	1	4	8	64	24	4	196,608	246	
19	(1-1-3-1)	1	1	1	3	1	4	8	4	13824	4	7,077,888	282	
20	(1-1-1-3)	1	1	1	1	3	4	8	4	24	64	196,608	67	
										Totals	77,377,536	3,491		

Key points:

1. Templates define patterns of system behavior and are identified by execution of the model, not superimposed on the model.
2. Increasing the scope of execution enables the developer to derive new templates.
3. When the increased scope produces no new interactions, the developer may have confidence that the templates sufficiently capture all behaviors of the model.

4.3.2.6.2 Pattern analysis: The universal templates indicate whether all events are in a template or identify the individual events that are not part of a template. As shown in Figure 4.78, before Template 19 was implemented, the second object was not addressed by a sensor and subsequent action and so the associated events were not part of a template. Figure 4.79 illustrates the associated graphic, after applying Template 19 as a warning. Similarly, Templates 20 through 22 were derived, and illustrated in Figures 4.80 through 4.82.

Finally, Figure 4.83, with one object, but two sensors, two processors, and two actors illustrates a case where one of the actors did not attempt an effect on the object, the associated sensing, processing, and acting functions were not linked to the object of interest, and so each of these templates were identified as isolated function and indicated as such in the model output.

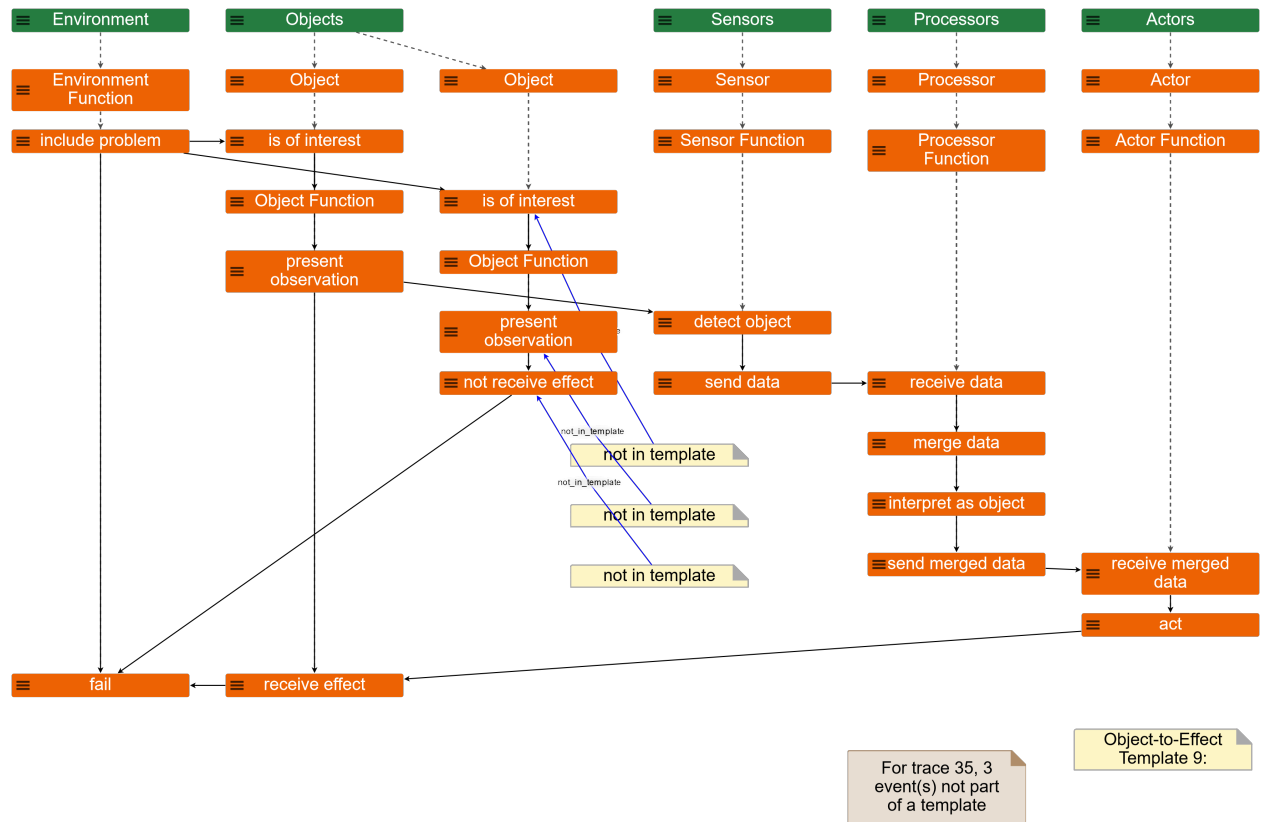


Figure 4.78. Object-to-Effect Model analysis, two objects, one sensor, one processor, and one actor, trace 35 of 72. Note that a valid sequence of events satisfies Template 9, but the second object is isolated from the activity of the sensors, processors, and actors.

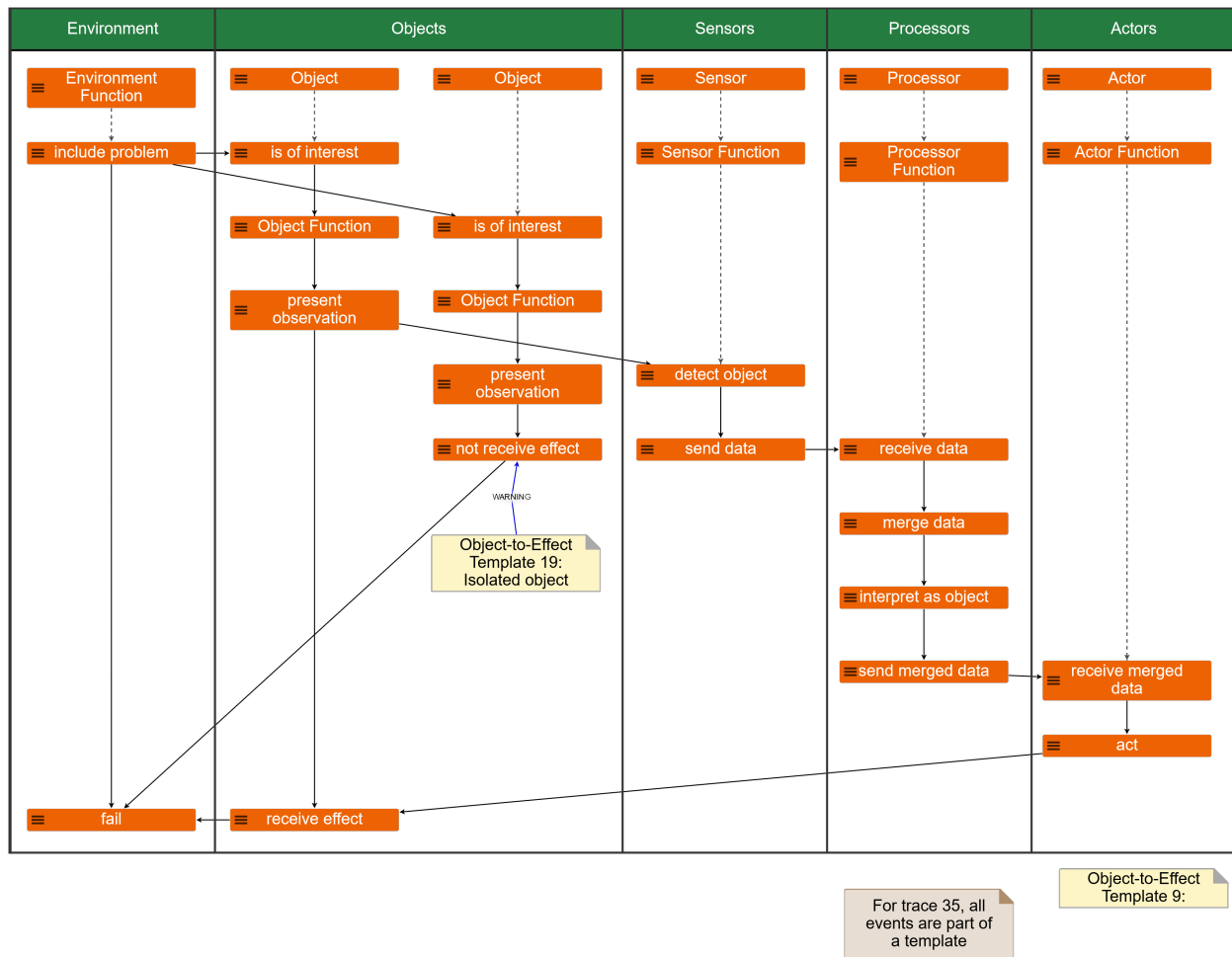


Figure 4.79. Object-to-Effect Model analysis two objects, one sensor, one processor, and one actor, trace 35 of 72. Execution accomplished with Template 19, indicating a warning of the isolated object, such that there exists no preceding action or non-action upon the second object.

MP code segment 4.58: Template 19.

```

/* Template 19: Object not processed */
/**/
COORDINATE $is2:  $$EVENT SUCH THAT
                  ( ($is2.It OR $is2.nIt ) )

DO COORDINATE $of2:  $$EVENT SUCH THAT ( $of2.isObjectFunction
                  AND $is2 BEFORE $of2 )

DO COORDINATE $show2:  $$EVENT SUCH THAT
                  ( $show2 IN $of2
                  AND ( $show2.Pr OR $show2.nPr ) )

DO COORDINATE $effect2:  $$EVENT SUCH THAT
                  ( $effect2 IN $of2
                  AND ( $effect2.Eff OR $effect2.nEff )
                  AND $show2 PRECEDES $effect2 )
DO IF (#($$EVENT) PRECEDES $effect2 == 1) THEN
    ADD SAY("Object-to-Effect Template 19: Isolated object")
        WARNING $effect2;
    $is2.inTemplate := true;
    $show2.inTemplate := true;
    $effect2.inTemplate := true;

FI; OD; OD; OD; OD;

```

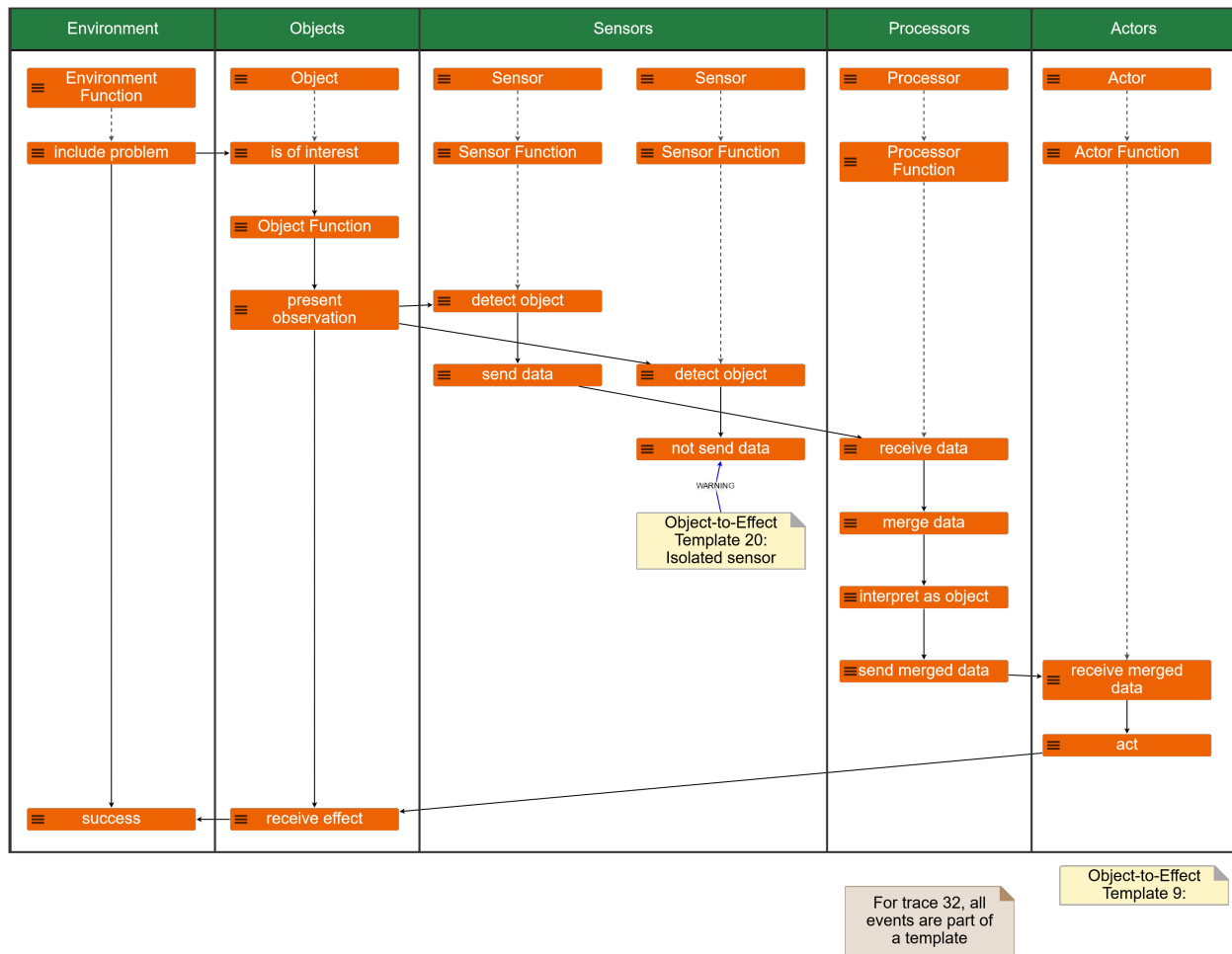


Figure 4.80. Object-to-Effect Model analysis trace 32 of 70 at scope of (1-2-1-1), illustrating an isolated sensor such that the data does not support an action on the object.

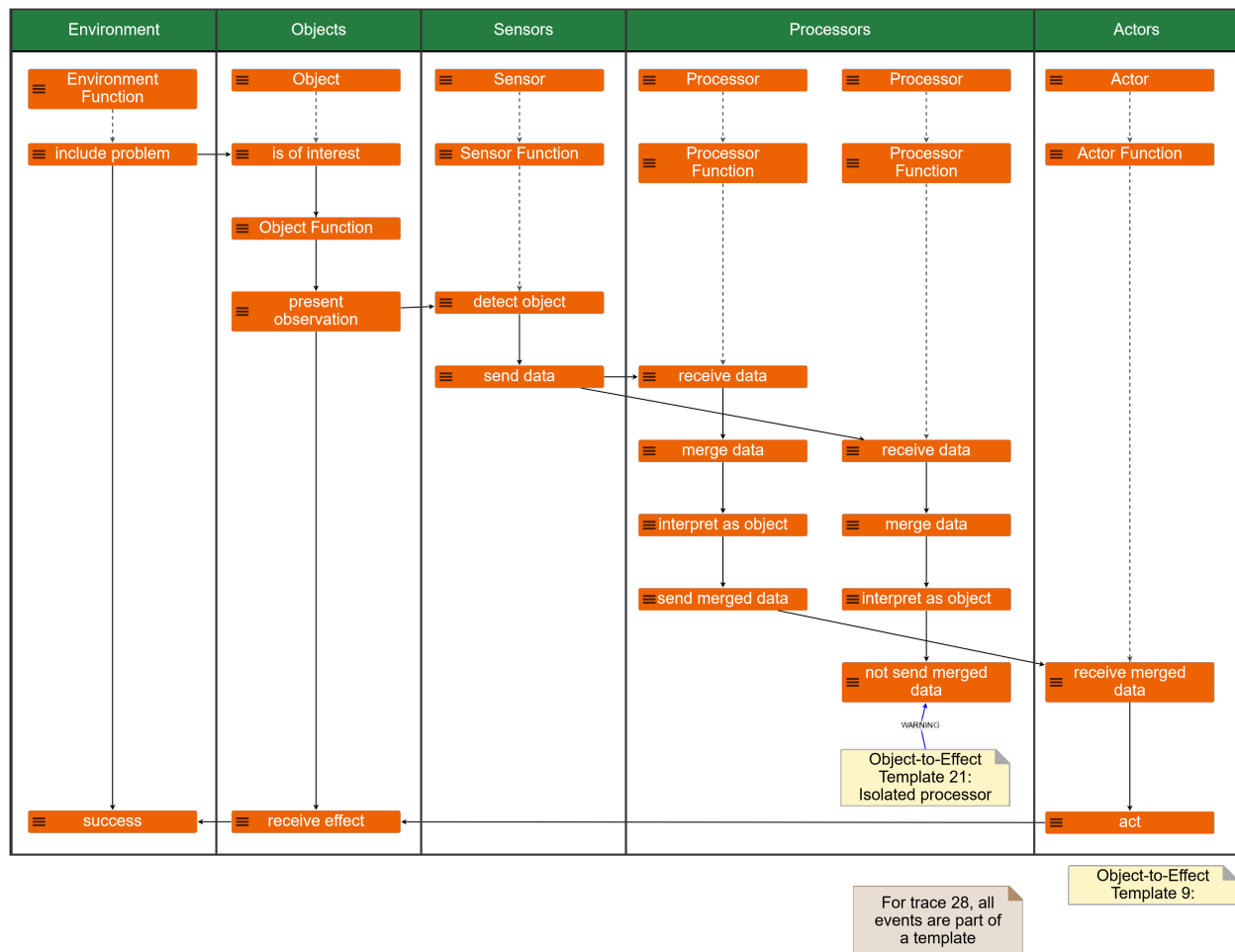


Figure 4.81. Object-to-Effect Model analysis trace 28 of 66 at scope of (1-1-2-1), illustrating an isolated processor such that the merged data does not support an action on the object.

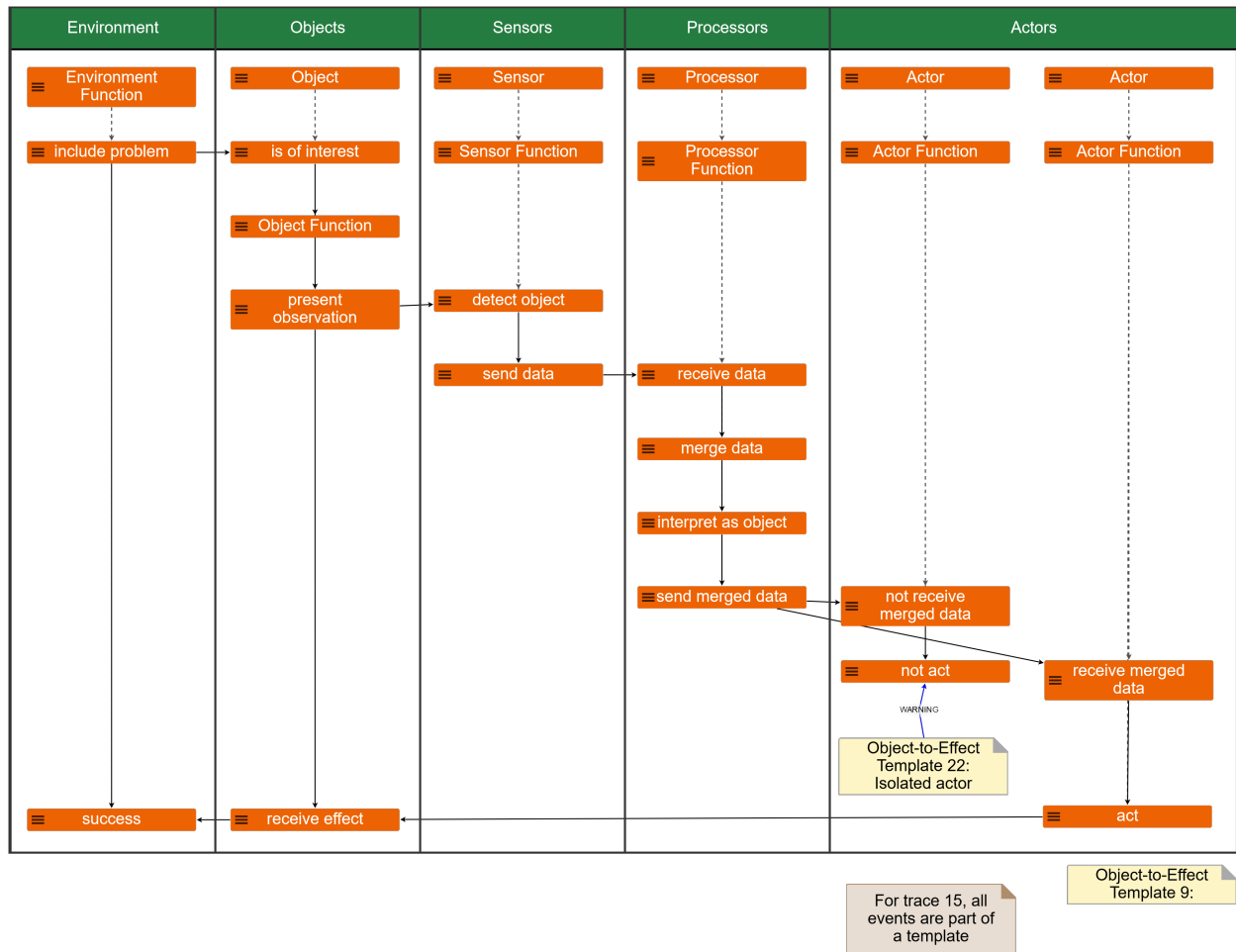


Figure 4.82. Object-to-Effect Model analysis trace 15 of 30 at scope of (1-1-1-2), illustrating an isolated action such that the object is not acted upon.

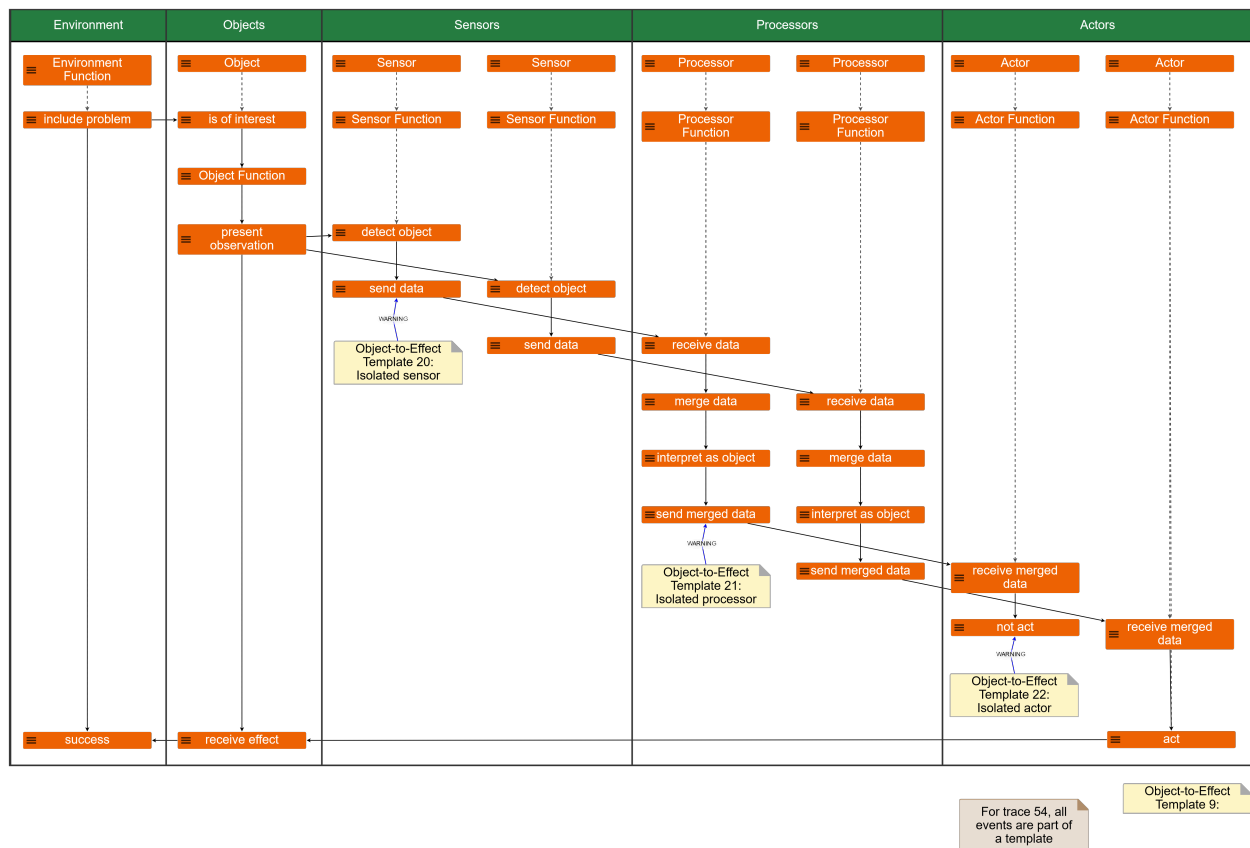


Figure 4.83. Object-to-Effect Model analysis trace 54 of 130 at scope of (1-2-2-2), illustrating an isolated sensor, processor, and actor.

4.3.2.6.3 The probability of each trace: These calculations were enabled by deriving a Bayesian belief network from the behavior model topology, and then aligning the constraints from the behavior model to this belief network. As illustrated in Figure 4.84, each node A through AE represents an event, and each node has an option of two alternatives, with the exception of the interpretation, node H, that has three alternatives: interpret as an object of interest, not an object of interest, or unknown (i.e., hostile, friendly, or unknown). Figure 4.85 illustrates the network, with conditional probabilities associated with each node, with the results indicated in Table 4.27. Table 4.28 summarizes the outcomes of the model.

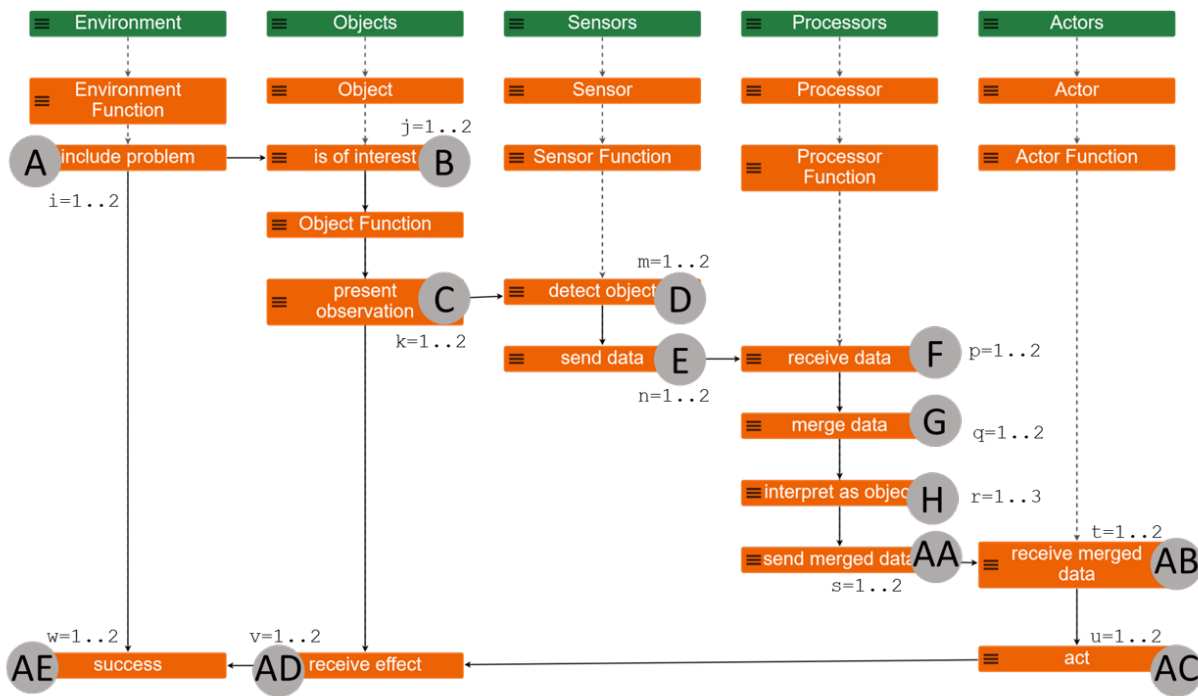


Figure 4.84. Object-to-Effect Model probability.

Table 4.27. Object-to-Effect Model probability calculation at scope of (1-1-1-1), given an equal probability of each alternative event.

Trace	Ref	P(trace)	i	P(a)	j	P(b)	k	P(c)	m	P(d)	n	P(e)	p	P(f)	q	r	P(h)	s	P(aa)	t	P(ab)	u	P(ac)	v	P(ad)	w	P(ae)	Template	
1	7	0.00391	1	0.500	1	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	1	1.000	1	0.500	1	0.500	2	1.000	2	1.000	1	1.000	T1: Not an object of interest
2	15	0.00391	1	0.500	1	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	1	1.000	1	0.500	2	0.500	2	1.000	2	1.000	1	1.000	T2: Non-object, merged data not received
3	31	0.00781	1	0.500	1	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	1	1.000	2	0.500	2	1.000	2	1.000	2	1.000	1	1.000	T3: Non-object, merged data not sent
4	191	0.01563	1	0.500	1	1.000	1	0.500	1	0.500	1	0.500	1	0.500	2	0.500	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	1	1.000	T4: Non-object, interpreted as unknown
5	383	0.03125	1	0.500	1	1.000	1	0.500	1	0.500	1	0.500	2	0.500	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	1	1.000	T5: Non-object, data not received
6	767	0.0625	1	0.500	1	1.000	1	0.500	1	0.500	2	0.500	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	1	1.000	T6: Non-object, data not sent
7	1535	0.125	1	0.500	1	1.000	1	0.500	2	0.500	2	1.000	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	1	1.000	T7: Non-object, not detected
8	3071	0.25	1	0.500	1	1.000	2	0.500	2	1.000	2	1.000	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	1	1.000	T8: Non-object, not observable
9	9249	0.00098	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	1.000	T9: Object receives effect
10	9252	0.00098	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	2	1.000	1	0.500	1	0.500	1	0.500	2	0.500	2	1.000	T10: Object acted upon without effect
11	9256	0.00195	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	2	1.000	1	0.500	1	0.500	2	0.500	2	1.000	2	1.000	T11: Object, but no action taken
12	9264	0.00391	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	2	1.000	1	0.500	2	0.500	2	1.000	2	1.000	2	1.000	T12: Object, merged data not received
13	9280	0.00781	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	1	0.500	2	1.000	2	0.500	2	1.000	2	1.000	2	1.000	2	1.000	T13: Object, merged data not sent
14	9408	0.01563	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	1	0.500	2	0.500	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	2	1.000	T14: Object interpreted as unknown
15	9600	0.03125	2	0.500	2	1.000	1	0.500	1	0.500	1	0.500	2	0.500	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	2	1.000	T15: Object, data not received
16	9584	0.0625	2	0.500	2	1.000	1	0.500	1	0.500	2	0.500	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	2	1.000	T16: Object, data not sent
17	10752	0.125	2	0.500	2	1.000	1	0.500	2	0.500	2	1.000	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	2	1.000	T17: Object not detected
18	12288	0.25	2	0.500	2	1.000	2	0.500	2	1.000	2	1.000	2	1.000	2	1.000	3	1.000	2	1.000	2	1.000	2	1.000	2	1.000	2	1.000	T18: Object not observable
Total:		1.0																											

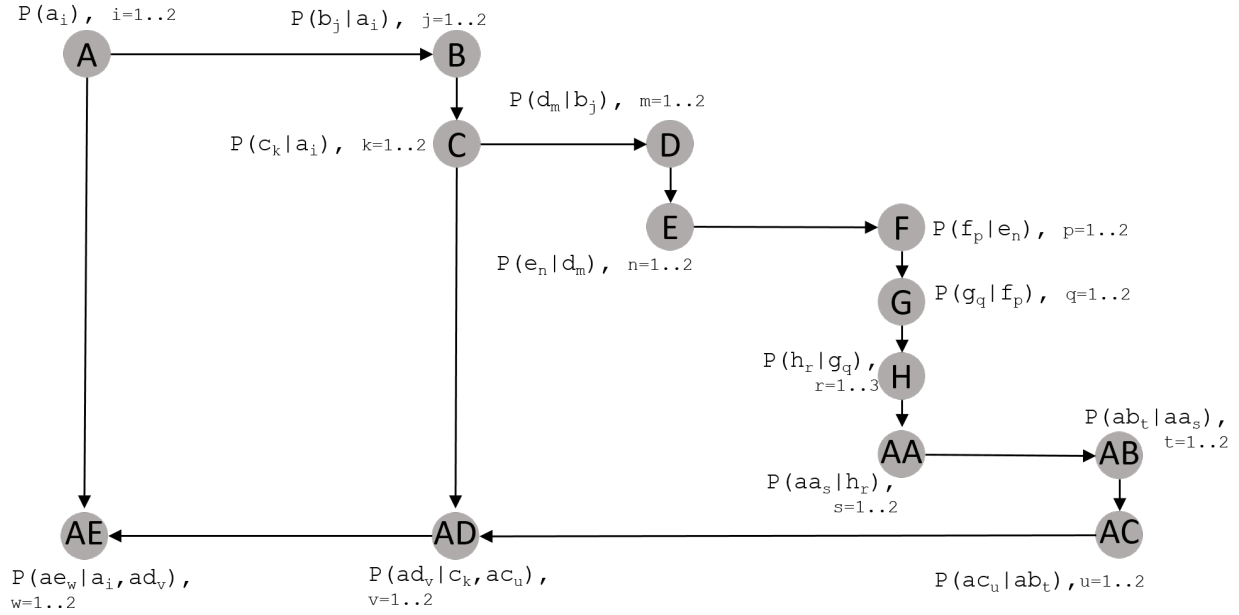


Figure 4.85. Object-to-Effect Model probability belief network.

Table 4.28. Object-to-Effect Model outcomes, given equal probability of occurrence for each event alternative.

Outcome	Value
Probability of success:	0.5010
Probability of failure:	0.4990
Probability of non-object of interest ID:	0.0156
Probability of object of interest ID :	0.0156
Probability of unknown ID :	0.9688

Improving model outcomes: By adjusting the assumptions used for the probability network, the developer may assess a means to improve the model outcomes. Iterations of this step can lead to developing requirements for each allocated subsystem so that the outcomes may be predicted and bounded. The following lists an example of adjusted parameters, followed by Table 4.29, listing the potentially improved outcomes. These may serve as system of system requirements, as allocated to each of the event functions.

Adjusted assumptions for an improved Object-to-Effect Model include the following:

- The object presents an observation 90% of the time,
- If observation is presented, then the sensor detects the object 99.9% of the time,
- If detected, then the sensor sends data 99.9% of the time,
- If the data is sent, then the processor receives the data 99.9% of the time,
- If the data is received, then merge the data 99.9% of the time,
- If the data is merged, then correctly interpret the object of interest or not an object of interest 99.9% of the time,
- If the merged data is sent from the processor to the actor, then the actor receives the track 99.9% of the time,
- If interpreted as an object of interest, and the actor receives the merged data, then act 90% of the time,
- If an action is taken, then have an effect 99% of the time

Table 4.29. Object-to-Effect Model improved outcomes, given goals of the probability of occurrence for event alternatives.

Outcome	Value
Probability of success:	0.8989
Probability of failure:	0.1011
Probability of non-object of interest ID:	0.4482
Probability of object of interest ID :	0.4482
Probability of unknown ID :	0.1036

4.3.2.6.4 The DSM: For the Object-to-Effect Model at scope of (1-1-1-1) and (2-1-1-1) the DSM matrices are listed in Tables 4.30 and 4.31, respectively. As expanded upon in Appendix C, analysis of these models indicated that the increased scope adds interactions among the events. The resulting output, derived by the MATLAB[®] script is as follows:

```
Sum of DSM edges from objectToEffectv2scope1111 is 123
Sum of DSM edges from objectToEffectv2scope2111 is 125
i = 10, j = 36: scope (2111) has relationship from ...
    include_problem to is_not_of_interest not shared by ...
    execution at scope (1111)
i = 41, j = 38: scope (2111) has relationship from ...
    receive_effect to fail not shared by execution at scope (1111)
```

These distinctions are expected since an additional template was derived and implemented as the scope was increased from (1-1-1-1) to (2-1-1-1). In this case, since the execution at scope (2111) has an additional object, the occurrence of “including a problem” and an object “not of interest” occurs since the second object is of interest. Similarly when two objects of interest do not receive an effect, the environment interprets that scenario as a failure.

Key points:

1. The DSM provides a means to view all interactions of the model in a single depiction.
2. Analytic comparison of the DSM at increasing scope provides a means to identify new interactions.
3. The DSM serves as a means to verify that the developer can identify all interactions associated with the model.

Table 4.30. Object-to-Effect Model DSM at scope of (1-1-1-1-1).

Event name	index	Environment	Objects	Sensors	Processors	Actors	Environment_Function	not_include_problem	success	Object	is_not_of_interest	Object_Function	present_observation	not_receive_effect	Sensor	Sensor_Function	detect_object	send_data	Processor	Processor_Function	receive_data	merge_data	interpret_as_not_object	send_merged_data	Actor	Actor_Function	receive_merged_data	not_act	not_receive_merged_data	not_send_merged_data	not_merge_data	interpret_as_unknown	not_receive_data	not_send_data	not_detect_object	not_present_observation	include_problem	is_of_interest	receive_effect	interpret_as_object	act	fail
Environment	(1)																																									
Objects	(2)									18																																
Sensors	(3)																																									
Processors	(4)																																									
Actors	(5)																																									
Environment_Function	(6)							8	9																																	
not_include_problem	(7)								8																																	
success	(8)																																									
Object	(9)																																									
is_not_of_interest	(10)																																									
Object_Function	(11)																																									
present_observation	(12)																																									
not_receive_effect	(13)																																									
Sensor	(14)																																									
Sensor_Function	(15)																																									
detect_object	(16)																																									
send_data	(17)																																									
Processor	(18)																																									
Processor_Function	(19)																																									
receive_data	(20)																																									
merge_data	(21)																																									
interpret_as_not_object	(22)																																									
send_merged_data	(23)																																									
Actor	(24)																																									
Actor_Function	(25)																																									
receive_merged_data	(26)																																									
not_act	(27)																																									
not_receive_merged_data	(28)																																									

Table 4.31. Object-to-Effect Model DSM at scope of (2-1-1-1), illustrating two interactions not shared by the model execution at (1-1-1-1), as highlighted in yellow.

[illegible]

4.3.2.7 Object-to-Effect Model re-use

As discussed previously, the simplest form of re-use of the model is a word-for-word replacement from an abstract-level model to a specific application-level model. However, having defined the functions, the developer may now employ these functions in new and various ways to achieve application-level models with different forms from the original abstraction.

The following examples include system-of-systems model and an automotive braking model.

4.3.2.7.1 Distributed system of system applications: Applications may include the same object with variations system-types with multiple functions to achieve some effect on the object of interest. Consider the following example of coordinated systems, with sensors on each platform, a single processor on one of these platforms, and an actor on one of the platforms:

- System 1 (2 ea): Sensor function
- System 2 (1 ea): Sensor and processor functions
- System 3 (1 ea): Sensor and actor functions

Employing the following MP code, and the remainder of the model to be identical, the developer can quickly develop the results illustrated in Figure 4.86.

```
ROOT External_system: { + <1> Object + }  
    Object: ( is_not_of_interest | is_of_interest )  
        Object_Function;  
  
ROOT System_Type_1: { + <2> System_1 + }  
    System_1: Sensor_Function;  
  
ROOT System_Type_2: { + <1> System_2 + }  
    System_2: {Sensor_Function, Processor_Function};  
  
ROOT System_Type_3: { + <1> System_3 + }  
    System_3: {Sensor_Function, Actor_Function};
```

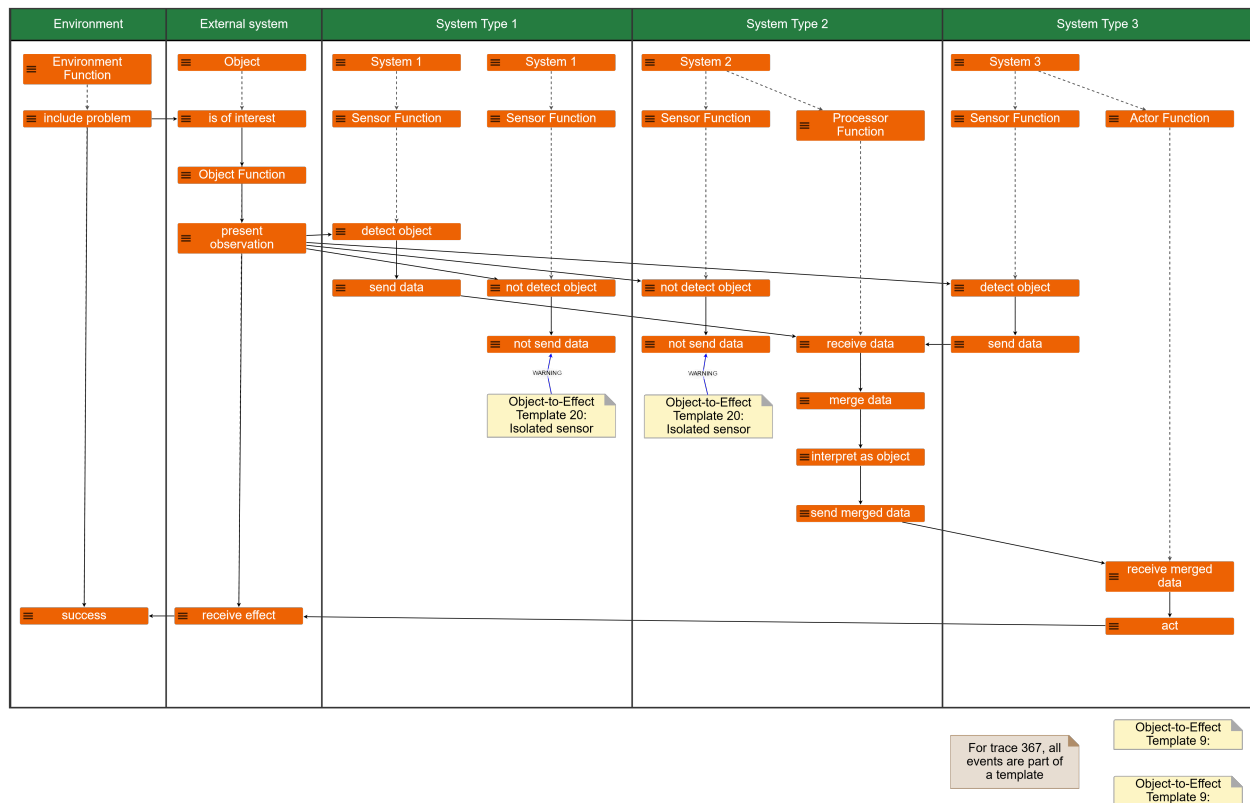


Figure 4.86. Object-to-Effect Model re-use of a system of systems with distributed sensors across three types of platforms. Notice that for this trace (367 of 1760 total traces), two of four sensors effectively contributed data to the processor. Also, two paths resulted in the object of interest receiving an effect, thereby satisfying Template 9.

4.3.2.7.2 Networked Automotive Braking Application: An additional application includes a concept of multiple vehicles with the same functions, but able to coordinate processing so that multiple sensors from each vehicle contribute to the processor input. Consider the following potential approach, with adjustments to the abstraction-level model as follows:

- Environment condition: Excessive closure speed
- Multiple vehicles with the following functions:
 - Own vehicle is or is not too close to neighbor vehicle
 - Object function *same as abstraction-level*
 - Sensor function *same as abstraction-level*
 - Processor function
 - * interpret as too close
 - * interpret as not too close
 - * interpret as unknown
 - Actor function to apply brakes

Employing the following MP code, the developer may employ the following model, as illustrated in Figure 4.87.

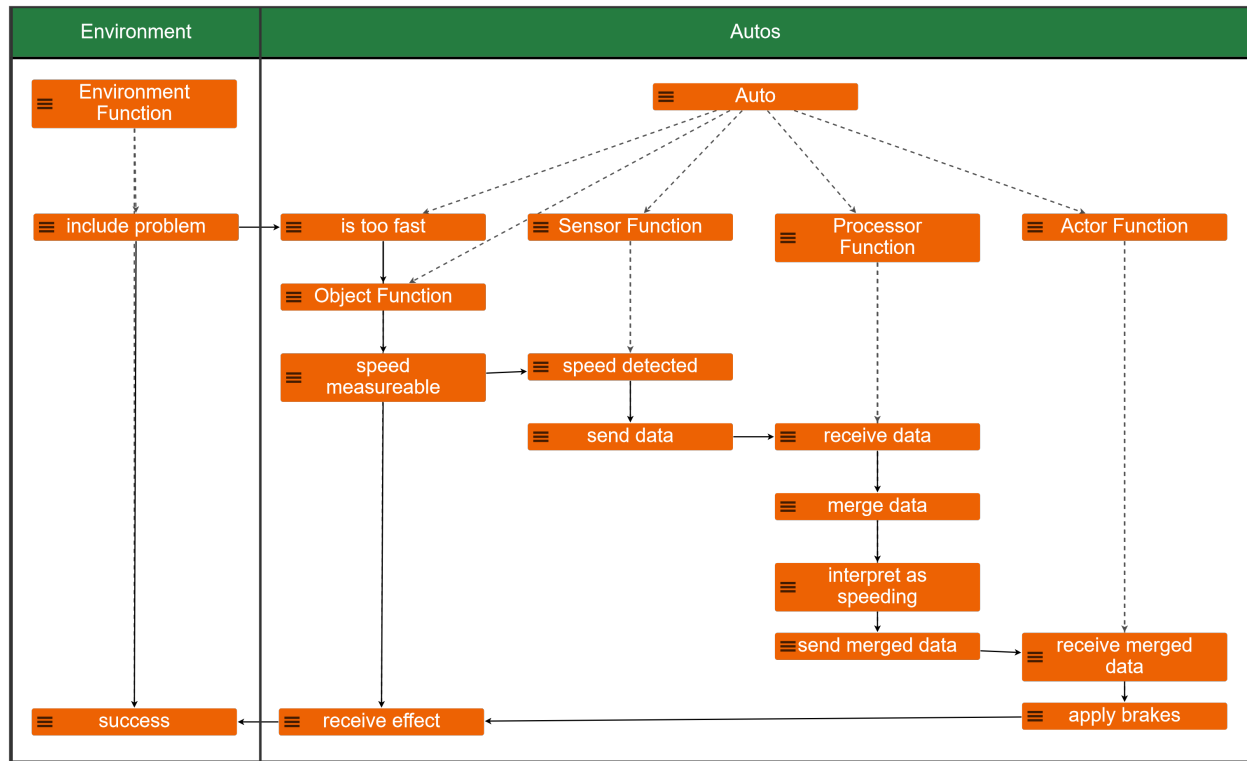
```

ROOT Autos: { + <1> Auto + };
Auto: { ( is_not_too_fast | is_too_fast )
        Object_Function,
        Sensor_Function, Processor_Function, Actor_Function };

```

4.3.2.8 Implications of the Object-to-Effect Model

The functions associated with sensors, processors, and actors enable a response to an object of interest. Developing and analyzing the behaviors of the Object-to-Effect Model provided an extensive means to have confidence in the underlying architecture, formally defined by the schema. The interactions and constraints establish a set of formal requirements of the functions to be allocated to systems that constitute the SoS. These well-defined functions may then be attributed to systems in many creative ways, enabling improved performance and reliability of the system under development.



For trace 9, all events are part of a template

Object-to-Effect Template 9:

Figure 4.87. Vehicle sensors model, derived from the Object-to-Effect abstraction-level model with sensors and processors networked among neighboring vehicles.

4.4 Summary of Findings

Researching patterns within behavior model architectures facilitated several findings, as listed below:

- The model description, including the events, the hierarchical and temporal relationships, coordination, and constraints provides a formal definition of the model that needs to be reflected in the fielded system in order to constrain the behavior of that system.
- Interrogating the model for patterns of behavior enables the developer to categorize types of outcomes and to simplify interpretation of the results.
- The developer can determine whether each event in all traces fit within a pattern, with full confidence that none were missed.
- The probability of occurrence of any particular trace was demonstrated using a Bayes model [168] for conditional probability. The developer assigns conditional probability for each given event that must be consistent with the constraints of the model. Deriving the probability of occurrence of each valid trace within the schema was derived for the Decision Model and Object-to-Effect Model.
- The Design Structure Matrix (DSM) is readily produced by executing the model, and inspection of this diagram provides a means to identify all possible interactions of the model within one view-point.
- The DSM facilitates extended analysis including clustering algorithms that identify regions of close interaction among events, supporting a means to incorporate modular segments within the system design.
- Applications include the development of a behavior model methodology to evaluate combinations of patterns of a System of Systems (SoS), with dependence upon capturing key interactions among multiple systems.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:

Summary

5.1 Discussion

The objective of a system of systems developer extends the traditional systems engineering approaches of requirements definition, functional analysis, and design synthesis in order to attain objectives not possible by a single system [38], [64], [69].

The developer of a system of systems needs to ensure that all possible behaviors of the design have been thoroughly investigated and verified, prior to product release to the end-user. Therefore, the developer faces a risk that inherent or latent characteristics may cause unexpected or unwanted outcomes in the system of systems. The design needs to exhibit desired behaviors to perform a particular function and also constrain undesired behaviors, avoiding hazardous or damaging outcomes.

Any designed system must rely on a precise description in order to capture all plausible outcomes and resulting behaviors. Ambiguity in the system description exposes the system to potential failure modes that are not noticed by the developer. Unfortunately, common systems engineering practices allow a level of ambiguity in the architecture description.

This dissertation has sought to equip the SoS developer by providing a systematic means to identify patterns that are inherent in the system description. Employing a formal definition of the model forces a complete definition, driving out ambiguity to the level of detail that is within the model. When executed, the methodology identifies patterns inherent in the design of the architecture. This identification enables a means to reuse successful patterns and to avoid failed or hazardous patterns. Failure and success is defined within the context of the system operational employment, and so this interpretation becomes important to capture, and is done through using consistent semantics that follow a governed ontology. The associated templates may then be available for future use within a repository available to system developers and architects.

Summary of the Dissertation Contribution:

This dissertation contributed to the body of knowledge in Systems Engineering by developing a novel means to identify behavior patterns in system of systems architectures. The associated methodology has the following features:

- The methodology emphasized a means to derive the core behaviors needed to accomplish a desired outcome of the system under development. Related behaviors were identified as functions that are then be allocated to each constituent system. These functions implement the integral interactions among the systems, users, and environment.
- The methodology delineated a taxonomy for the model constraints, establishing a formal definition of system requirements, derived interactively with the Monterey Phoenix (MP) Firebird analyzer. This taxonomy includes logical, design, definition, and simplification types of constraints.
- The methodology outlined a means to derive both positive and negative behavior patterns from the set of possible outcomes using templates that captured the topology of the model and the semantics of each critical activity within the model.
- The methodology outlined a means to analyze and interpret the results by interactively deriving the templates, verifying that all interactions were captured by a template, calculating the probability of each potential outcome, and evaluating all interactions by tabulating the Design Structure Matrix (DSM).
- The methodology demonstrated a means to re-use the model for other problems of interest by implementing a repository of models and associated templates.
- Example problems illustrated the methodology, including 1) a human decision model of a leader and subordinate interchange during a stressful situation, and 2) a system of systems means to employ an effect on an object of interest.
- Step-by-step instructions documented how the reader may employ the methodology to further research.

- The method, schemata, scripts, and applications were made available for follow-on research through this published dissertation.

5.2 Conclusions

This dissertation developed a methodology that is intended to derive inherent patterns of a behavior model. Understanding patterns inherent in system architecture helps the developer to identify valid paths of execution and to resolve large sets of potentially undesirable execution paths of the system. By interacting with the model, the developer can refine a set of constraints that serve as a set of requirements. Types of constraints include logical, design, simplification, and definition.

The approach relied upon the use of Monterey Phoenix (MP), a light-weight formal method that ensures that the model completely captures all possible sets of alternative events, within the scope of execution of the model. As a formal method, the model specification is based upon a mathematical representation, thereby ensuring completeness and supporting future means of analyzing the model.

Demonstration of the methodology centered on two abstraction-level models (1) a human decision model, and (2) a designed systems of systems of a distributed operational mission. The former example outlined the steps taken by a leader and subordinate(s) that contribute to a critical decision. The corresponding outcomes, capture documented failure modes and also illustrate the critical points at which a successful outcome could have been achieved. The latter example of a designed system of systems developed the description of the functions associated with sensors, processors, and actors in order to have an effect upon an object of interest. These functions could then be allocated to systems within the model, with very simple changes of the code. Applications included a model of interacting vehicles producing during an emergency braking procedure. These examples illustrate that a significant depth of analysis can be accomplished with a model defined at a relatively high level of abstraction. This approach lends itself to model reuse of the full schema or of segments of the schema, as governed by a consistent use of semantics within a governed ontology structure.

5.3 Suggestions for Further Research

This section outlines areas of further research that are relevant to the concepts of behavior modeling and pattern identification. These recommendations include capabilities within Monterey Phoenix (MP) and the concepts of a repository necessary to capture the behavior patterns and thus be available for reuse.

5.3.1 Recommended Capabilities for Monterey Phoenix (MP)

Recommendations for MP include some automated or assisted means to improve computational efficiency, align the probability belief network to the model constraints and coordination, derive the DSM, and develop interfaces to SysML.

5.3.1.1 Computational efficiency

The methodology outlined within this dissertation employed sequential steps to help the developer to detail a behavior model in a sequential fashion. As indicated by the examples, it is easy to develop a model with thousands of possible outcomes, and the subsequent template search query becomes relatively intensive. Faster execution may be to apply the model constraints earlier in the schema, as close as possible to the event descriptions. In this way, subsequent processing does not need to query branches of the model that have been eliminated as invalid. The developer may do this manually in the schema, however, it may be beneficial for others to interpret the model if the original structure remains sequential. Therefore, it is recommended that prior to execution, that MP would conduct a step to align the schema to improve efficiency without disrupting the human interface with the schema.

5.3.1.2 Probability structures

This dissertation illustrated that a Bayesian belief structure may be developed in order to calculate the probability of occurrence of each trace, as outlined in Appendix B and employed in the examples of Section 4.3.1 and 4.3.2. The model coordination and constraints change the underlying Bayes belief network, and so implementing probability calculations depends upon alignment of the constraints and coordination with the belief network. Domains for applying probability calculations include each alternative event, each sequence of events, each trace in an iteration, each iteration in the schema, and each trace in the schema. Consistency of the axioms of probability need to apply across each of these domains, as outlined in Appendix B. Therefore, it is recommended that MP provide a means to automatically update the assignment of the probability structure and to give the developer an optional display with which to edit the probability assignments of each conditional event.

5.3.1.3 The DSM

As illustrated in the dissertation, the DSM enables the developer to investigate all of the relationships among all events within a single diagram. This gives an indication of highly used interfaces and lightly used interfaces. Furthermore, implementing a cluster algorithm, as illustrated in Figure C

also provides a means to align modules more efficiently, such that those events that have a higher level of exchanges are located closer to each other within the system. Therefore, it is recommended that MP develop a means to produce the DSM, the associated cluster analysis, and a listing of all event relationships, separated by category (inclusion, precedence, and user_defined).

5.3.1.4 Interfaces between MP and SysML

As indicated in Section 2.1.4, MP and SysML were derived from a completely different starting point. The former, employing formal methods, relies on a precise language based in propositional and predicate logic that forms a mathematical composition of the model; this specification enables structured means for verification and further analysis. The latter is based on graphical notation, and is dependent upon the developer to ensure completeness, and is not readily adaptable to an automated means of verification. Nonetheless, the graphical representations of SysML and the associated relevance to MBSE poses a significant advantage over traditional systems engineering, and as modeling tools develop, applying a formal method definition that forms an underlying foundation to MBSE tools would significantly increase the utility of the systems engineering practice.

From SysML to MP: Jarraya [180] proposed a means to translate SysML activity diagrams into a formal language, thereby employing a form of first order logic to the diagram. This construct may be adapted to MP syntax, with a means to expand the application to the other behavior related diagrams: sequence diagrams, state machine diagrams, and use-cases. This introduces an immediate challenge, because in order to translate these diagrams into a formal language, the developer must resolve potential discrepancies within the behaviors described in the model. Logical discrepancies are of particular concern, but the developer also needs to consider design consistency as well. As outlined in this dissertation, formally defined constraints are a key aspect of behavior development, and these constraints need to be translated into the SysML model.

From MP to SysML: Execution of MP in accordance with the methodology outlined in this dissertation would enable the developer to evolve a set of constraints that may be thought of as requirements for the system. Relating this description, along with the event relationships of precedence, inclusion, and user-defined taxonomy, within a SysML profile could at least capture the model in a way that is organized within the MBSE model.

5.3.2 Pattern Repository and Management

One of the purposes for developing patterns in SoS architectures is to enable reuse of the model for future projects. Re-use may be accomplished by applying an entire model to multiple disciplines, as the aviation mishap has similarities to a medical crisis. Also, the developer may wish to apply the model to a new project, as the OODA loop applies to many detailed models. Inherent in the concept of reuse is a consistent use of semantics that follows an identical ontology across all models that employ models within some repository. Structuring and controlling that repository reflects an additional area of work that would be essential to the systems engineering community.

APPENDIX A: Behavior Models

A.1 Decision Model

A.1.1 Decision Model Graphic Output

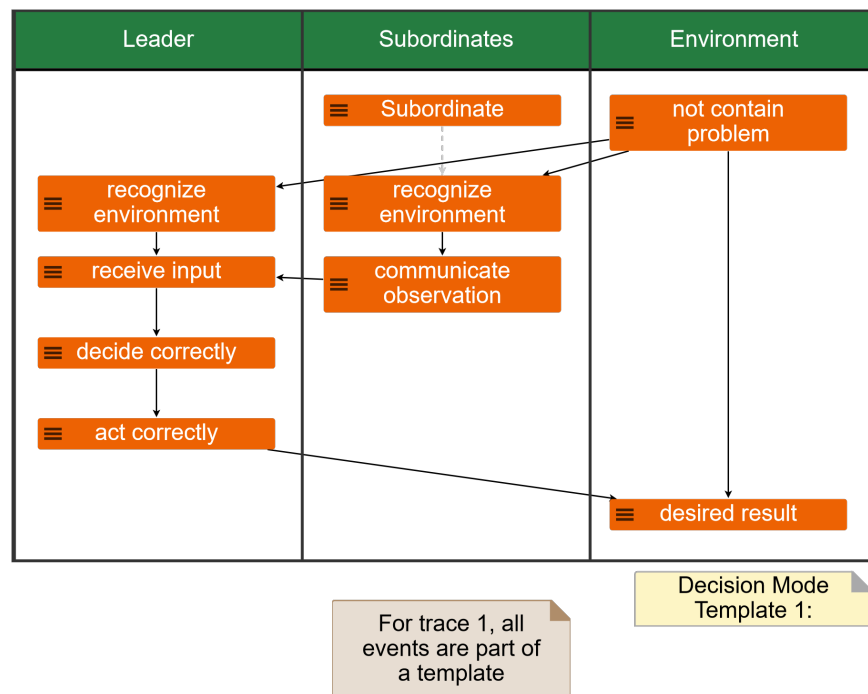


Figure A.1. Decision Model, scope 1, trace 1 of 12.

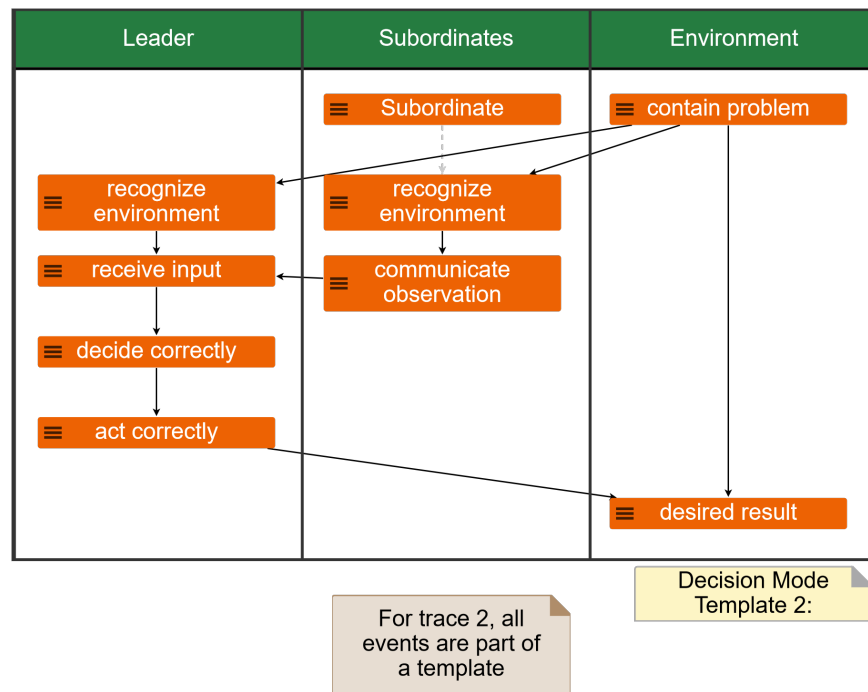


Figure A.2. Decision Model, scope 1, trace 2 of 12.

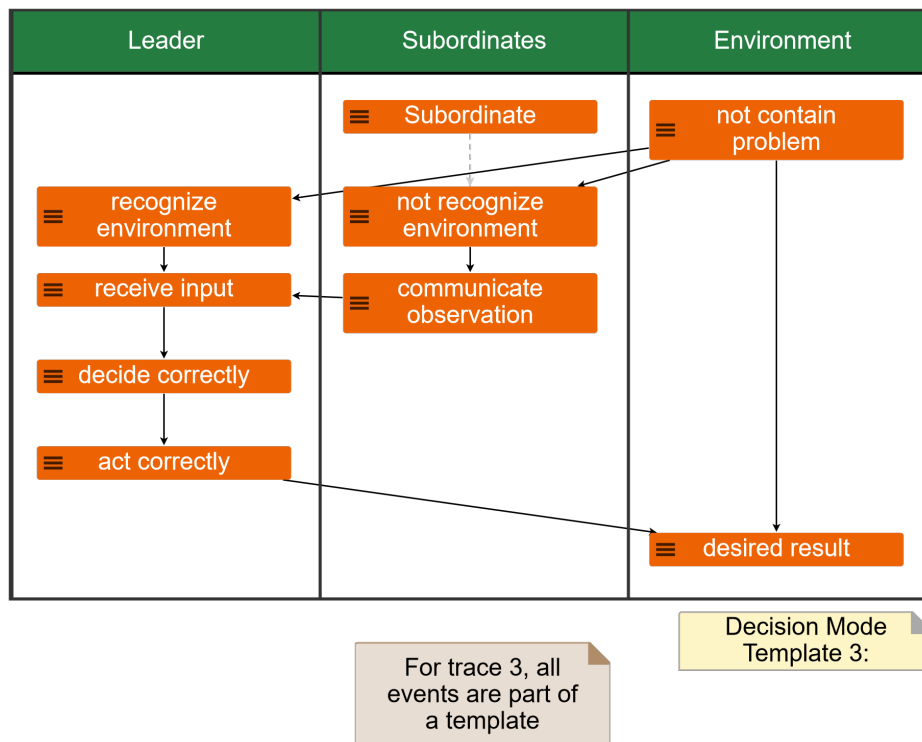


Figure A.3. Decision Model, scope 1, trace 3 of 12.

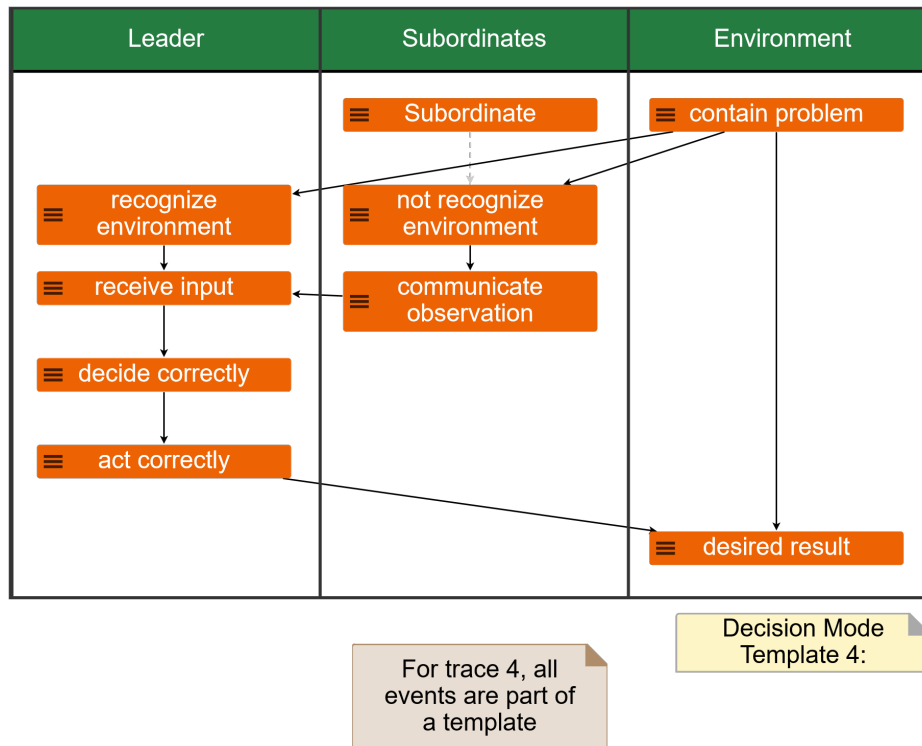


Figure A.4. Decision Model, scope 1, trace 4 of 12.

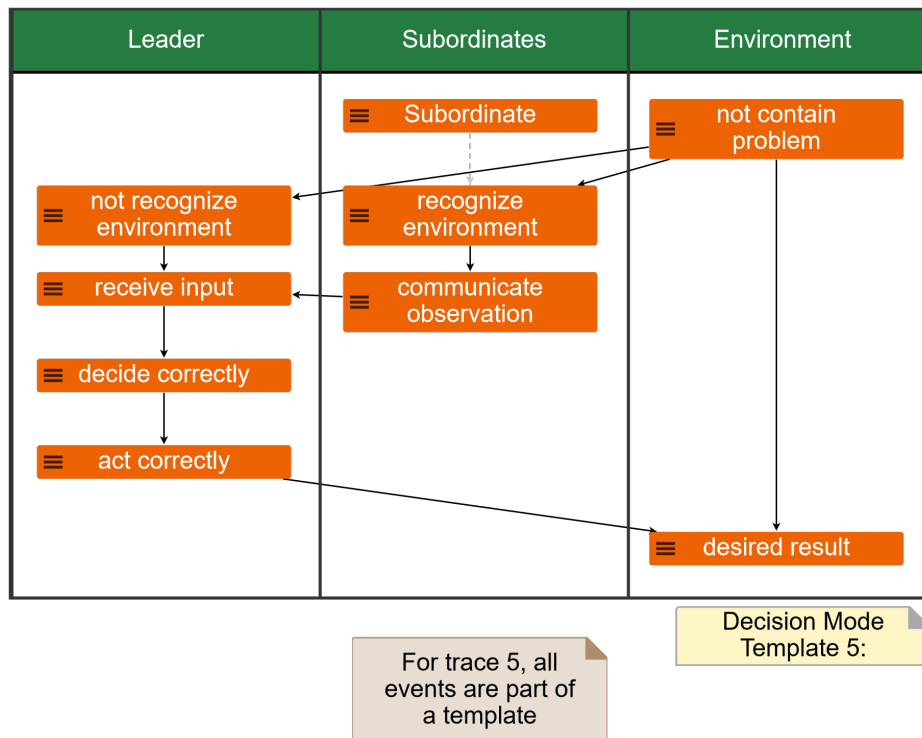


Figure A.5. Decision Model, scope 1, trace 5 of 12.

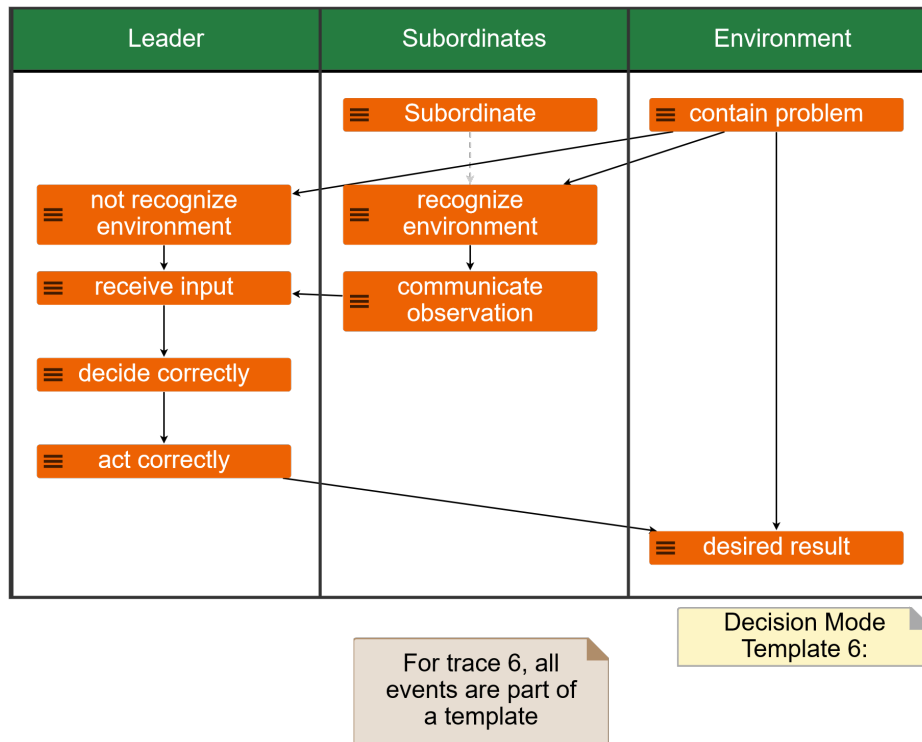


Figure A.6. Decision Model, scope 1, trace 6 of 12.

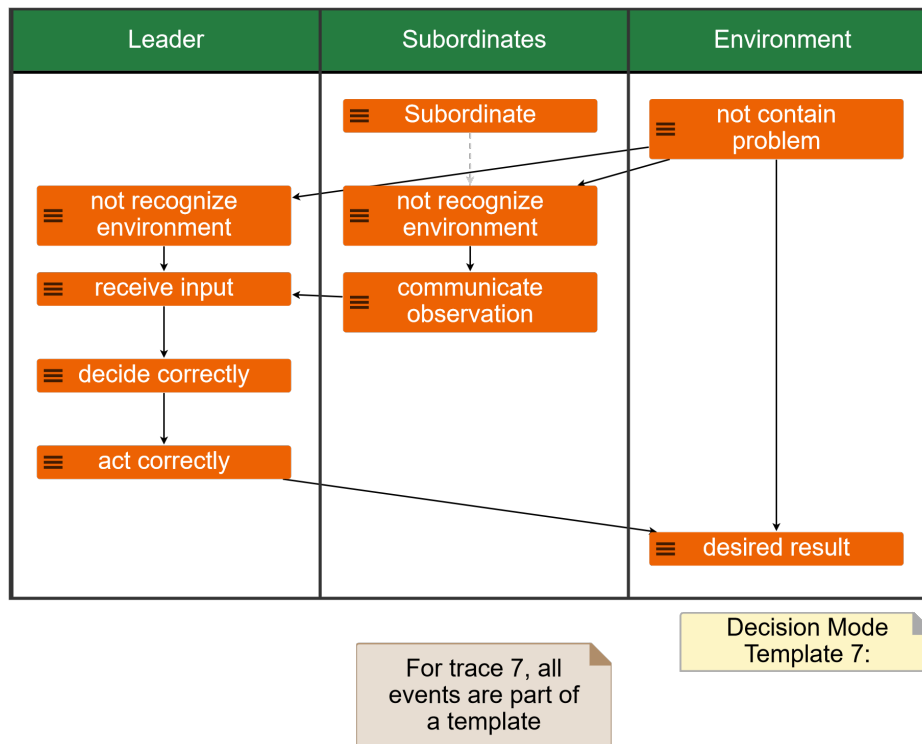


Figure A.7. Decision Model, scope 1, trace 7 of 12.

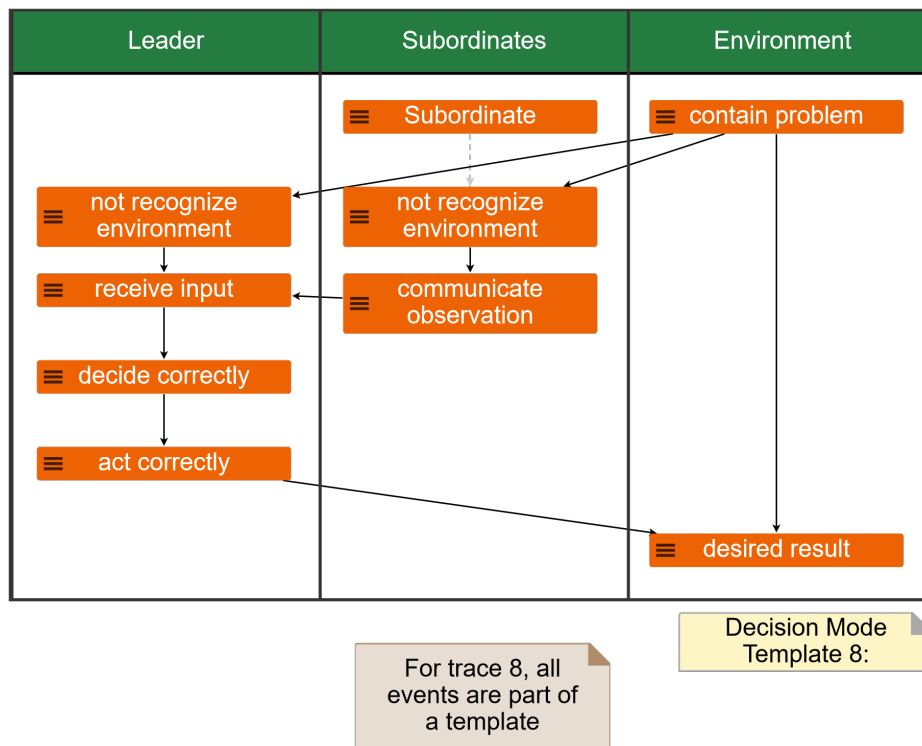


Figure A.8. Decision Model, scope 1, trace 8 of 12.

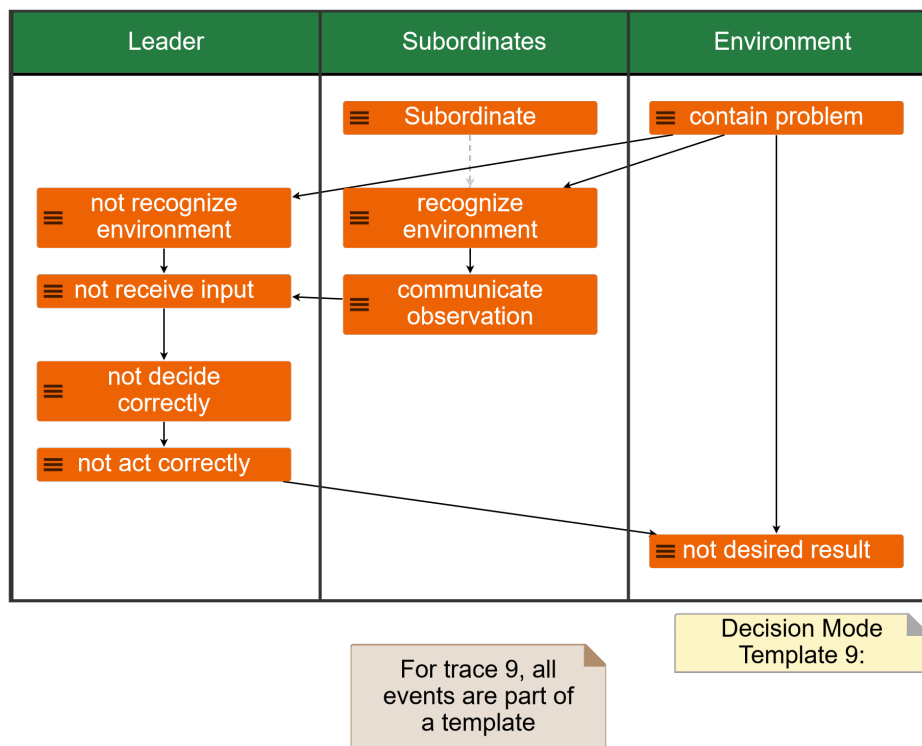


Figure A.9. Decision Model, scope 1, trace 9 of 12.

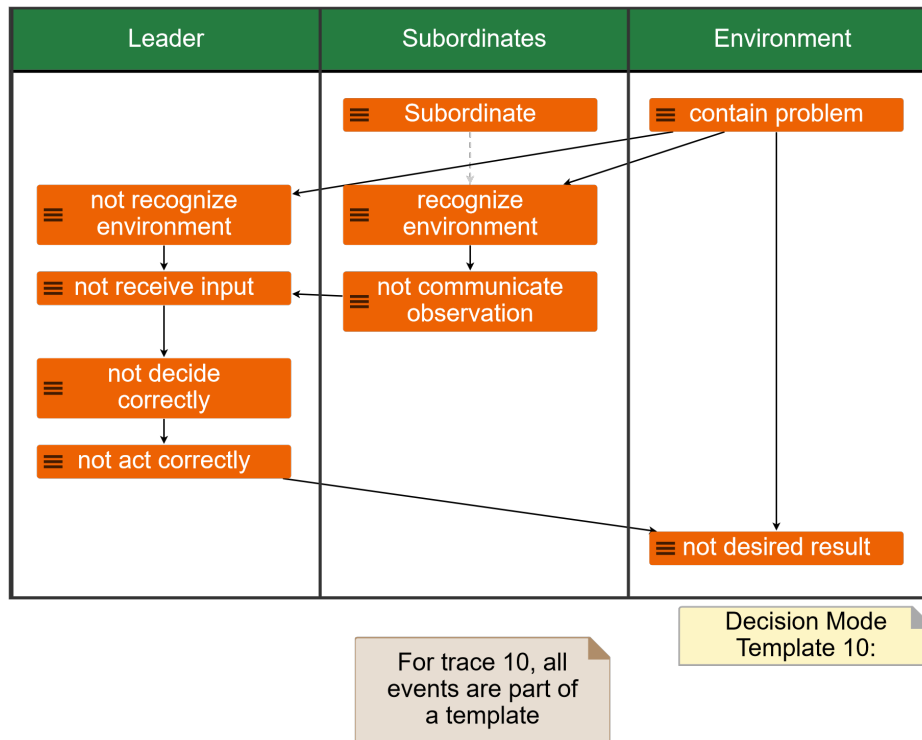


Figure A.10. Decision Model, scope 1, trace 10 of 12.

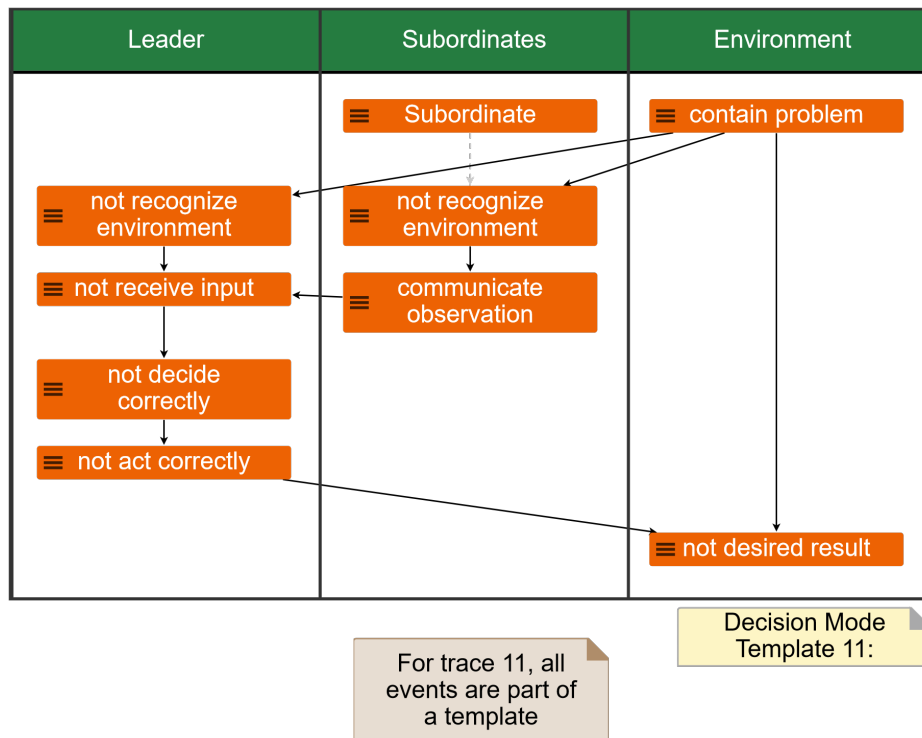


Figure A.11. Decision Model, scope 1, trace 11 of 12.

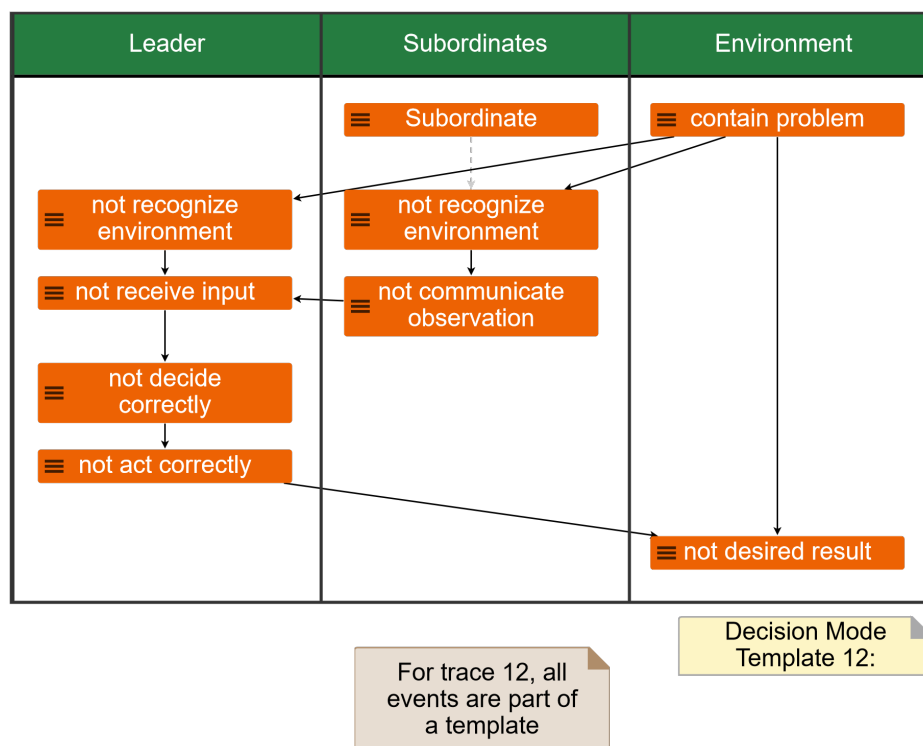


Figure A.12. Decision Model, scope 1, trace 12 of 12.

A.1.2 Decision Model MP Code Listing

```
/*
Decision Model Schema version 22
Developed by John Quartuccio
This schema illustrates a Decision Mode relevant to a hierarchical
leadership structure such that the leadership context may
disregard correct perception from a subordinate team member.

Motivation derived from similarities between aviation and medical
procedures, derived from Syed (2015).

06 Feb 2017 v9
updated with edits from Dr. Auguston and Dr. Giammarco

09 May 2017 v10 Updated to eliminate constraint 4 in order to
maintain probability of total outcomes equal to one, while given
initial probabilities for the leader's perception, subordinate's
perception, and contains_problem state. This results in twelve
templates at scope 1.

16 Jun 2017 v11 updated as final version for publications
13 JUL 2018 v12 simplified template search patterns
19 JUL 2018 v13 added abstact pattern searches
11 AUG 2018 v14 added OODA loop events and semantics
14 OCT 2018 v15 syntax changes
30 OCT 2018 v16 OODA loop updates
04 NOV 2018 v17 Attributes test case
18 NOV 2018 v18
30 JUN 2019 v19 Attribute-based templates implemented
01 JUL 2019 v20 Includes pattern segments
30 AUG 2019 v21 Includes checks for events in a template
02 SEP 2019 v21
25 OCT 2019 v22 Finalized for dissertation
*/

SCHEMA decisionModelv22

/*****EVENT BEHAVIORS*****/
/*****
ROOT Leader:
    ( recognize_environment
      | not_recognize_environment )

    ( receive_input | not_receive_input )

    ( decide_correctly | not_decide_correctly )

    ( act_correctly | not_act_correctly);

ROOT Subordinates:  {+ <1> Subordinate +};

    Subordinate:
        ( recognize_environment
          | not_recognize_environment )

        ( communicate_observation
          | not_communicate_observation );

ROOT Environment:
    ( not_contain_problem | contain_problem )

    ( desired_result | not_desired_result );
```



```

/*****COORDINATION*****/
/*****Interaction 1: The problem state precedes the perception */
/**/
COORDINATE $x: ( not_contain_problem
                | contain_problem )
DO COORDINATE
  $y: ( recognize_environment
        | not_recognize_environment )
DO ADD $x PRECEDES $y; OD; OD;

/****Interaction 2: Communication by the subordinate precedes */
/* the leadership interpretation of that communication */
/*
COORDINATE      $a: ( communicate_observation
                    | not_communicate_observation )

DO COORDINATE $b: ( receive_input
                    | not_receive_input )

DO ADD $a PRECEDES $b; OD; OD;

/* Optional conditional coordination
/**/
COORDINATE      $a: communicate_observation
DO COORDINATE $b: ( receive_input
                    | not_receive_input )
DO ADD $a PRECEDES $b; OD; OD;

COORDINATE      $a: ( communicate_observation
                    | not_communicate_observation )
DO COORDINATE $b: not_receive_input

DO ADD $a PRECEDES $b; OD; OD;

/****Interaction 3: An action leads an outcome */
/**/
COORDINATE $a: ( act_correctly
                | not_act_correctly),
  $b: ( desired_result
        | not_desired_result )
DO ADD $a PRECEDES $b; OD;

```

```

/*****CONSTRAINTS*****/
/*****Constraint 1. LOGICAL: If there is no contains_problem in */
/* the environment, then always have a successful outcome. */
/**/
ENSURE (#not_contain_problem FROM Environment == 1 ->
        #desired_result FROM Environment == 1);

/****Constraint 2. LOGICAL: If all subordinates do not */
/* communicate, then the leader receives no input. */
/**/
ENSURE (#not_communicate_observation FROM Subordinates -
        #Subordinate == 0 -> #not_receive_input FROM Leader == 1);

/****Constraint 3. SIMPLIFICATION: If the leader recognizes the*/
/* environment and does not receive input, then the leader */
/* makes a correct decision. */
/**/
ENSURE (#recognize_environment FROM Leader -
        #not_receive_input == 0 -> #decide_correctly == 1);

/****Constraint 4. SIMPLIFICATION: Not receiving an input leads*/
/* to an incorrect decision, and its corollary. */
/**/
ENSURE (#receive_input FROM Leader == 1 ->
        #decide_correctly FROM Leader == 1);
ENSURE (#not_receive_input FROM Leader == 1 ->
        #not_decide_correctly FROM Leader == 1);

/****Constraint 5. SIMPLIFICATION: A correct decision leads to */
/* a correct action, and its corollary. */
/**/
ENSURE (# decide_correctly == # act_correctly);
ENSURE (#not_decide_correctly == #not_act_correctly);

/****Constraint 6. DEFINITION: A correct action leads to a */
/* successful outcome, and its corollary. */
/**/
ENSURE(# act_correctly == #desired_result);
ENSURE(#not_act_correctly == #not_desired_result );

```

```

/*****
/*****Attributes Associated with each Event*****/
/*****
/*
/* Attribute definitions indicate as Boolean expressions, to be */
/* included in a repository, employing a defined ontology. */

ATTRIBUTES{bool is_authority, is_assistant, is_environ,
             favorable, unfavorable,
             Se, Ob, Or, De, Ac,
             nSe, nOb, nOr, nDe, nAc,
             Tx, Rx, Re, Int,
             nTx, nRx, nRe, nInt,
             Ex, nEx,
             fail, success;
};

/* Define attributes to identify whether events in a template */
ATTRIBUTES {bool inTemplate; number count, tracecount;};

/* Attribute assignments associated with each event, as written
   in the natural language of the subject matter. */
             Leader.is_authority := true;
             Leader.inTemplate  := true;
             Environment.is_environ := true;
             Environment.inTemplate := true;
             Subordinates.inTemplate := true;
COORDINATE $x: Subordinate DO $x.is_assistant := true; OD;

not_contain_problem :
    BUILD{ nEx:= true;   favorable:= true; };
contain_problem :
    BUILD{ Ex:= true; unfavorable:= true; };

recognize_environment :
    BUILD{ Ob:= true; Se:= true;   favorable := true; };
not_recognize_environment :
    BUILD{ nOb:= true; nSe:= true; unfavorable := true; };

receive_input :
    BUILD{ Rx:= true; Or:= true;   favorable:= true; };
not_receive_input :
    BUILD{ nRx:= true; nOr:= true; unfavorable:= true; };

decide_correctly :
    BUILD{ De:= true;   favorable:= true; };
not_decide_correctly :
    BUILD{ nDe:= true; unfavorable:= true; };

act_correctly :
    BUILD{ Ac:= true;   favorable:= true; };
not_act_correctly :
    BUILD{ nAc:= true; unfavorable:= true; };

communicate_observation :
    BUILD{ Tx:= true;   favorable:= true; };
not_communicate_observation :
    BUILD{ nTx:= true; unfavorable:= true; };

desired_result :
    BUILD{ success:= true;   favorable:= true; };
not_desired_result :
    BUILD{ fail:= true; unfavorable:= true; };

```

```

/*****
/*****Template Library*****/
/*****
/* This template library employs the attribute definitions from*/
/* the repository, and may be employed directly, without the */
/* natural language from any domain area. */
/*****

/*****
/* Decision Mode Template 1: Both leader and subordinate correctly perceive
there is no problem */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.nEx

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.Se
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 1:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 2: Both leader and subordinate
   correctly perceive a problem */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.Se
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)

DO
  SAY("Decision Mode Template 2:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 3: Subordinate perceives a problem that
   does not exist. */
/****/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.nEx

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.Se
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)

DO
  SAY("Decision Mode Template 3:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 4: Subordinate fails to perceive a real
   problem. */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.Se
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 4:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 5: Leader perceives a problem that does
   not exist. */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.nEx

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 5:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```



```

/*****/
/* Decision Mode Template 6: Leader incorrect perception, trusts
subordinate. */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 6:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 7: Incorrect perception, but no problem
exists. */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.nEx

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 7:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 8: Incorrect perception, but correct
decision. */
/**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.Rx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.De
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.Ac
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.success
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 8:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Decision Mode Template 9: Leader fails to consider
subordinate. */
**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.nRx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.nDe
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.nAc
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.Tx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.fail
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 9:"); MARK;
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Decision Mode Template 10: Subordinate fails to communicate
problem. */
**/
COORDINATE $authority:      $$EVENT SUCH THAT
                             $authority.is_authority

DO COORDINATE $assistant:   $$EVENT SUCH THAT
                             $assistant.is_assistant

DO COORDINATE $problem:     $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:    $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.nRx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.nDe
AND $receive BEFORE $decide)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.nAc
AND $decide BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.Se
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.nTx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:      $$EVENT SUCH THAT ($result.fail
AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 10:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate    := true;
  $decide.inTemplate     := true;
  $act.inTemplate        := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate   := true;
  $result.inTemplate     := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 11: Leader and Subordinate(s) have
   incorrect perception but with communication. */
/****/
COORDINATE $authority:    $$EVENT SUCH THAT
                           $authority.is_authority

DO COORDINATE $assistant:  $$EVENT SUCH THAT
                           $assistant.is_assistant

DO COORDINATE $problem:    $$EVENT SUCH THAT $problem.Ex

DO COORDINATE $perceive:   $$EVENT SUCH THAT ($perceive.nSe
                           AND $perceive IN $authority
                           AND $problem BEFORE $perceive)

DO COORDINATE $receive:    $$EVENT SUCH THAT ($receive.nRx
                           AND $perceive BEFORE $receive)

DO COORDINATE $decide:     $$EVENT SUCH THAT ($decide.nDe
                           AND $receive  BEFORE $decide)

DO COORDINATE $act:        $$EVENT SUCH THAT ($act.nAc
                           AND $decide   BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
                           AND $sub_perceive IN $assistant
                           AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:   $$EVENT SUCH THAT ($transmit.Tx
                           AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:     $$EVENT SUCH THAT ($result.fail
                           AND $act BEFORE $result)
DO
  SAY("Decision Mode Template 11:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****/
/* Decision Mode Template 12: All wrong. */
/**/
COORDINATE $authority:    $$EVENT SUCH THAT
                           $authority.is_authority

DO COORDINATE $assistant:  $$EVENT SUCH THAT
                           $assistant.is_assistant

DO COORDINATE $problem:    $$EVENT SUCH THAT  $problem.Ex

DO COORDINATE $perceive:   $$EVENT SUCH THAT ($perceive.nSe
AND $perceive IN $authority
AND $problem BEFORE $perceive)

DO COORDINATE $receive:    $$EVENT SUCH THAT ($receive.nRx
AND $perceive BEFORE $receive)

DO COORDINATE $decide:     $$EVENT SUCH THAT ($decide.nDe
AND $receive  BEFORE $decide)

DO COORDINATE $act:        $$EVENT SUCH THAT ($act.nAc
AND $decide   BEFORE $act)

DO COORDINATE $sub_perceive: $$EVENT SUCH THAT ($sub_perceive.nSe
AND $sub_perceive IN $assistant
AND $problem BEFORE $sub_perceive)

DO COORDINATE $transmit:   $$EVENT SUCH THAT ($transmit.nTx
AND $sub_perceive BEFORE $transmit)

DO COORDINATE $result:     $$EVENT SUCH THAT ($result.fail
AND $act BEFORE $result)

DO
  SAY("Decision Mode Template 12:");
  $authority.inTemplate := true;
  $assistant.inTemplate := true;
  $problem.inTemplate   := true;
  $perceive.inTemplate  := true;
  $receive.inTemplate   := true;
  $decide.inTemplate    := true;
  $act.inTemplate       := true;
  $sub_perceive.inTemplate := true;
  $transmit.inTemplate  := true;
  $result.inTemplate    := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Decision Mode Template 13. */
**/
COORDINATE $assistant1:  $$EVENT SUCH THAT
                           $assistant1.is_assistant

DO COORDINATE $assistant2:  $$EVENT SUCH THAT
                           ($assistant2.is_assistant
                           AND NOT
                           $assistant2.inTemplate )

DO COORDINATE $perceive1:  $$EVENT SUCH THAT (
                           ( $perceive1.Se OR $perceive1.nSe)
                           AND $perceive1 IN $assistant1 )

DO COORDINATE $perceive2:  $$EVENT SUCH THAT (
                           ($perceive2.Se OR $perceive2.nSe)
                           AND $perceive2 IN $assistant2 )

DO COORDINATE $transmit1:  $$EVENT SUCH THAT ($transmit1.Tx
                           AND $perceive1 PRECEDES $transmit1 )

DO COORDINATE $transmit2:  $$EVENT SUCH THAT ($transmit2.nTx
                           AND $perceive2 PRECEDES $transmit2 )

DO COORDINATE $receive:    $$EVENT SUCH THAT ($receive.Rx
                           AND $transmit1 PRECEDES $receive
                           AND $transmit2 PRECEDES $receive  )

DO
  ADD SAY("Decision Mode Template 13: nTx precedes Rx")
                                WARNING $transmit2;
  $assistant2.inTemplate      := true;
  $perceive2.inTemplate       := true;
  $transmit2.inTemplate        := true;
  MARK;

OD;  OD;  OD; OD; OD; OD; OD;

```



```

/*****
/* Decision Mode Template 14. */
**/
    COORDINATE $assistant1: $$EVENT SUCH THAT
                                $assistant1.is_assistant

DO COORDINATE $assistant2:    $$EVENT SUCH THAT
                                ( $assistant2.is_assistant
                                AND NOT $assistant2.inTemplate )

DO COORDINATE $perceive1:    $$EVENT SUCH THAT (
                                ($perceive1.Se OR $perceive1.nSe)
                                AND $perceive1 IN $assistant1 )

DO COORDINATE $perceive2:    $$EVENT SUCH THAT (
                                ($perceive2.Se OR $perceive2.nSe)
                                AND $perceive2 IN $assistant2 )

DO COORDINATE $transmit1:    $$EVENT SUCH THAT ($transmit1.Tx
                                AND $perceive1 PRECEDES $transmit1 )

DO COORDINATE $transmit2:    $$EVENT SUCH THAT ($transmit2.nTx
                                AND $perceive2 PRECEDES $transmit2 )

DO COORDINATE $receive:      $$EVENT SUCH THAT ($receive.Rx
                                AND $transmit1 PRECEDES $receive
                                AND NOT $transmit2 PRECEDES $receive )

DO
    ADD SAY("Decision Mode Template 14: Isolated " $assistant2)
                                WARNING $transmit2;

    $assistant2.inTemplate    := true;
    $perceive2.inTemplate     := true;
    $transmit2.inTemplate     := true;
    MARK;

OD;  OD;  OD; OD; OD; OD; OD;

/* End of Decision Mode Templates */

```

```

/*****
/*****General Template Library *****/
/*****
/* This template library employs the attribute definitions from*/
/* the repository, and may be employed directly, without the */
/* natural language from any domain area. */
/* */
/* These are general templates, applicable to all models that */
/* employ semantics that are consistent within the defined */
/* ontology. */
/* */
/*****
IF (0) THEN /* Set to one in Order to Implement Segments *****/
/*****Identifying Favorable and Unfavorable Events*****/
/*****

/**/
COORDINATE $parent: $$EVENT
DO COORDINATE $positiveEvents:  $$EVENT
    SUCH THAT  $positiveEvents.favorable
DO
    IF ($positiveEvents IN $parent) THEN
        SAY ("Positive event: " $parent " does " $positiveEvents);
FI; OD; OD;

/**/
COORDINATE $parent: $$EVENT
DO COORDINATE $negativeEvents:  $$EVENT
    SUCH THAT  $negativeEvents.unfavorable
DO
    IF ($negativeEvents IN $parent) THEN
        SAY ("Negative event: " $parent " does " $negativeEvents);
FI; OD; OD;

/* Identify the Send-Receive */
/**/
COORDINATE $transmit:  $$EVENT SUCH THAT $transmit.Tx
DO COORDINATE $receive:  $$EVENT SUCH THAT ($receive.Rx
    AND $transmit PRECEDES $receive)
DO
    SAY("Transmit-Receive: Tx:" $transmit ", Rx:" $receive );
OD; OD;

/* Identify a Relay, such that receive and transmit are in the */
/* same event */
/**/
COORDINATE $receive:  $$EVENT SUCH THAT $receive.Rx
DO COORDINATE $transmit:  $$EVENT SUCH THAT $transmit.Tx
DO
    IF ($transmit.Rx) THEN
        SAY("Relay found: both Rx and Tx in the same event "
            $transmit );
FI; OD; OD;

```

```

/* Identify orientation that depends upon reception */
/**/
COORDINATE $receive:    $$EVENT SUCH THAT $receive.Rx
DO COORDINATE $orient:  $$EVENT SUCH THAT $orient.Or
DO
  IF ($orient.Rx) THEN
    SAY("Orientation and reception in the same event: "
      $orient );
  FI; OD; OD;

/* Identify the OODA LOOP */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $orient:  $$EVENT SUCH THAT ($orient.Or
  AND $orient AFTER $observe)
DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.De
  AND $decide AFTER $orient)
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
  AND $act AFTER $decide)
DO
  SAY("OODA loop: " $observe ", " $orient ", " $decide ", "
    $act);
OD; OD; OD; OD;

/* Identify a Coordinated OODA LOOP */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $transmit:  $$EVENT SUCH THAT ($transmit.Tx
  AND $transmit AFTER $observe)
DO COORDINATE $receive:  $$EVENT SUCH THAT ($receive.Rx
  AND $receive AFTER $transmit)
DO COORDINATE $orient:  $$EVENT SUCH THAT ($orient.Or
  AND $orient AFTER $observe)
DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.De
  AND $decide AFTER $orient)
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
  AND $act AFTER $decide)
DO MARK;
SAY("Coordinated OODA loop: " $observe ", " $transmit ", "
  $receive ", " $orient ", "
  $decide ", " $act);
OD; OD; OD; OD; OD; OD;

/* Identify an uninformed negative action */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.nOb
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
  AND $act.unfavorable)
DO
  SAY("Uninformed Action: " $observe ", then " $act); MARK;
OD; OD;

/**/
FI; /* End of General Template Segment of Schema *****/

```

```

/*****
/*****GENERAL STATISTICS*****/
/*****
/*Mark the traces that have events not in a template*****/
**/
COORDINATE      $a:$SEVENT SUCH THAT NOT $a.inTemplate
  DO ADD SAY("not_in_template") not_in_template $a;
    GLOBAL.count += 1;
    count += 1;
    MARK;
  OD;
/*****
/* Statistic Reports for each trace *****/
**/
REPORT GeneralReport {};
IF (count > 0) THEN
  SAY("For trace " trace_id ", " count
      " event(s) not part of a template")
=>GeneralReport;
ELSE
  SAY("For trace " trace_id
      ", all events are part of a template")
=>GeneralReport;
FI;

IF(count > 0) THEN GLOBAL.tracecount += 1; FI;

SHOW  GeneralReport;
CLEAR GeneralReport;

/*****
/* Global Reports for the entire schema *****/
**/
GLOBAL
IF (GLOBAL.count > 0 ) THEN
  SAY("Summary: " GLOBAL.count " event(s) in " GLOBAL.tracecount
      " trace(s) are not part of a template, as marked")
    => GeneralReport;
ELSE
  SAY("Summary: All events in all traces are part of a template")
    => GeneralReport;
FI;
SHOW GeneralReport;

/*End of General Report Section*/

```

A.2 Object-to-Effect Model

A.2.1 Object-to-Effect Model Graphic Output

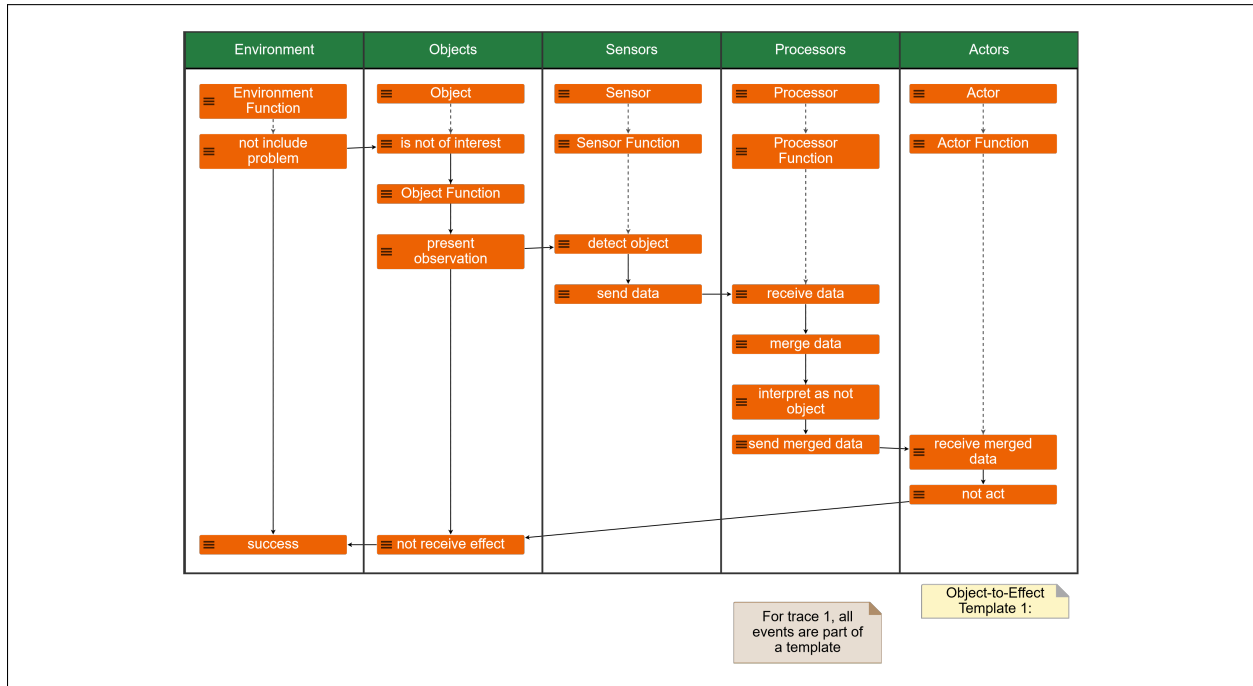


Figure A.13. Object-to-Effect Model, scope (1-1-1-1), trace 1 of 18, Template 1: Not an object of interest.

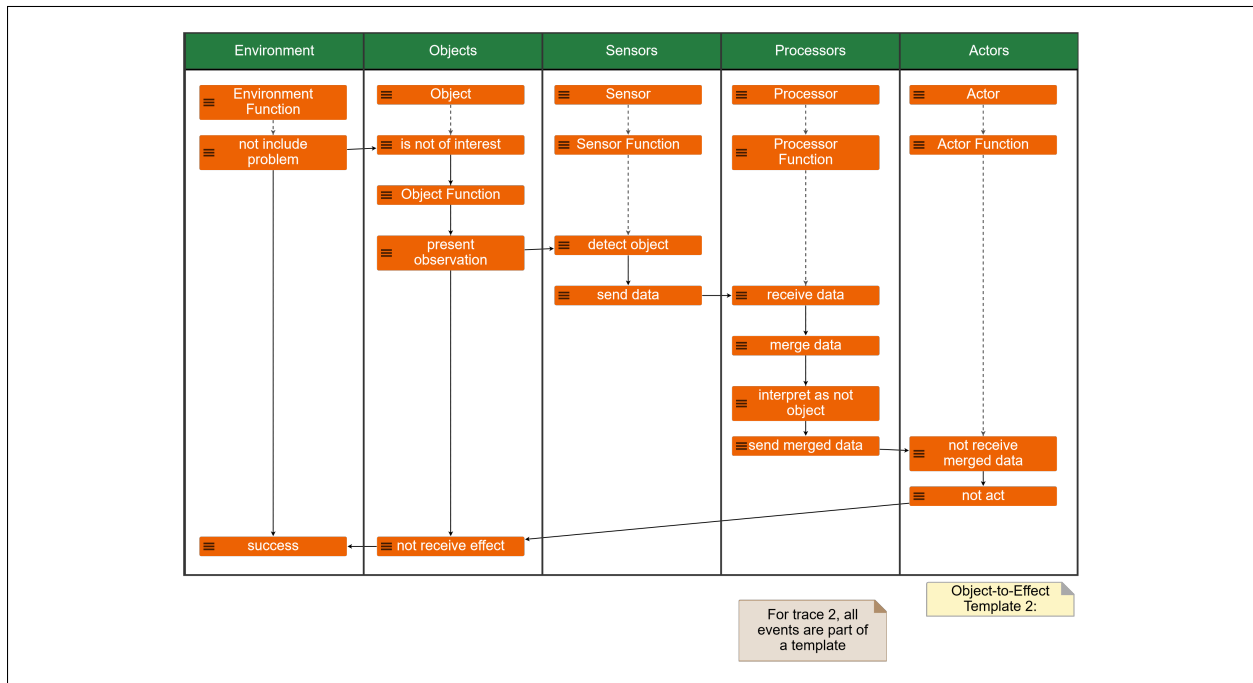


Figure A.14. Object-to-Effect Model, scope (1-1-1-1), trace 2 of 18, Template 2: Not an object of interest, merged data not received.

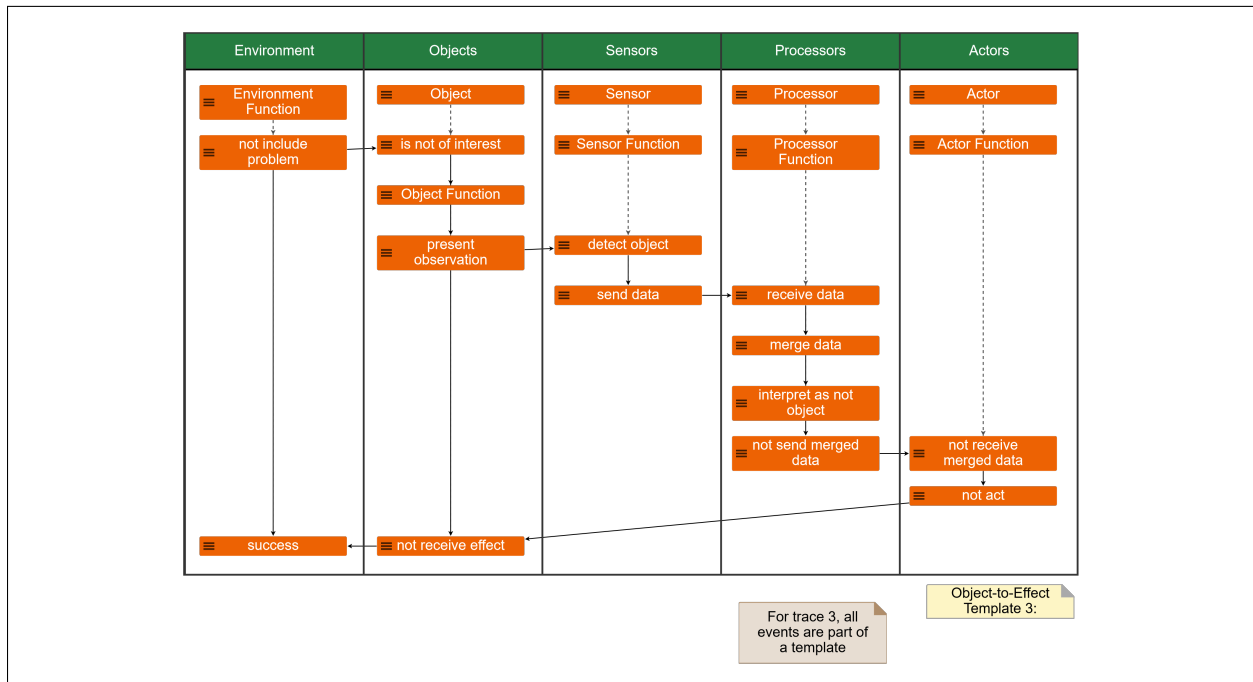


Figure A.15. Object-to-Effect Model, scope (1-1-1-1), trace 3 of 18, Template 3: Not an object of interest, merged data not sent.

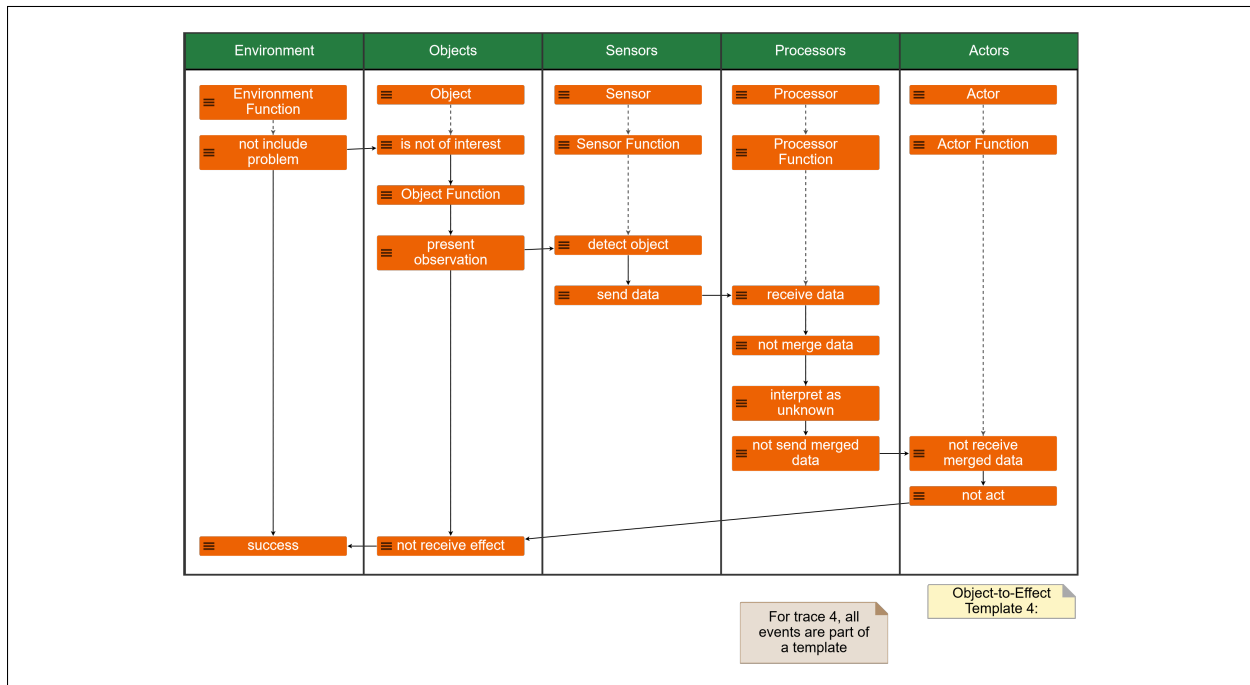


Figure A.16. Object-to-Effect Model, scope (1-1-1-1), trace 4 of 18, Template 4: Not an object of interest, interpreted as unknown.

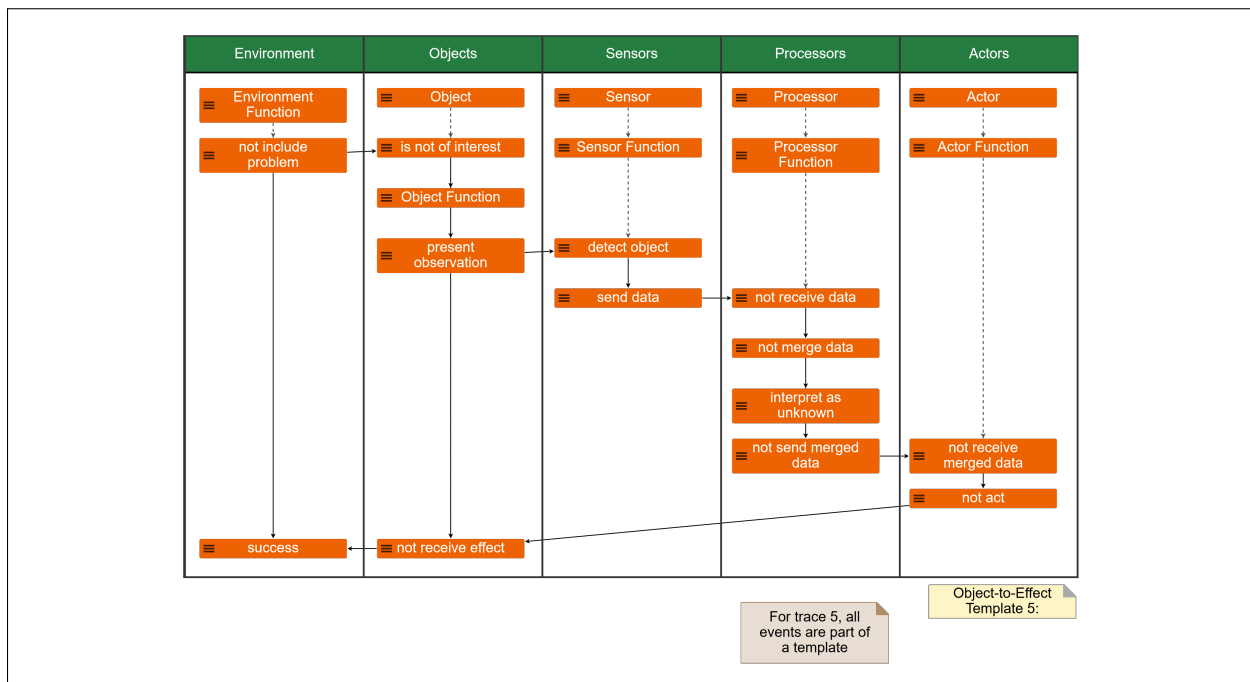


Figure A.17. Object-to-Effect Model, scope (1-1-1-1), trace 5 of 18, Template 5: Not an object of interest, data not received.

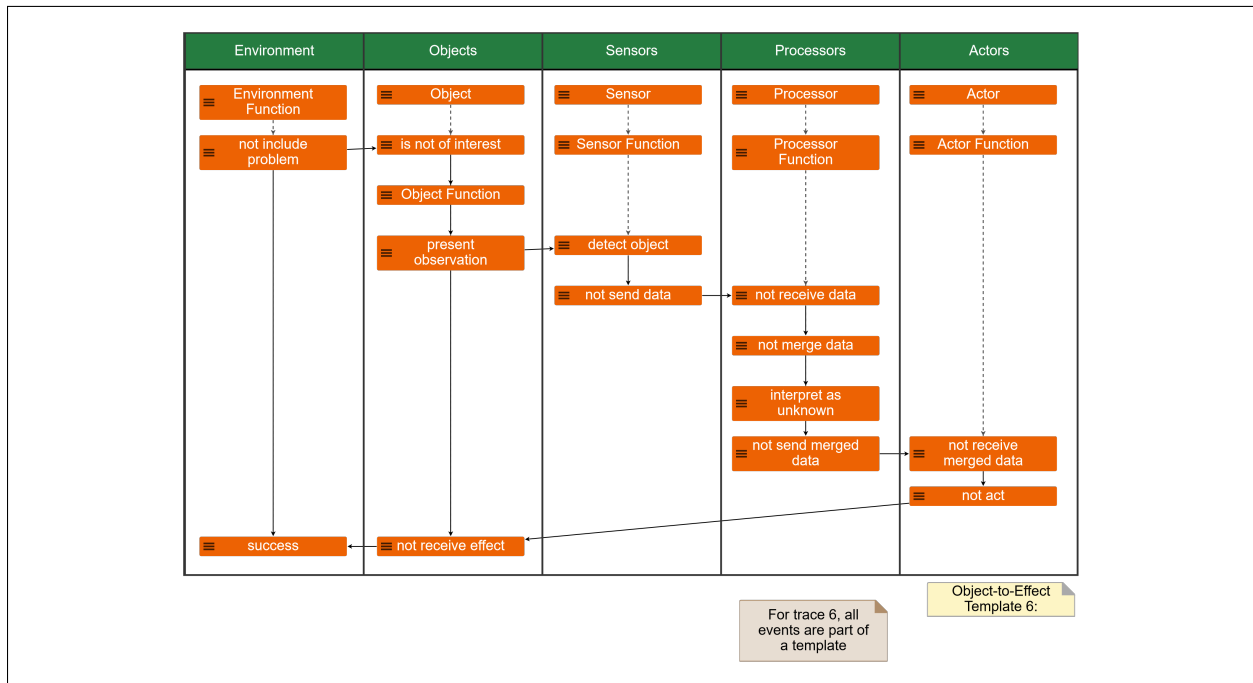


Figure A.18. Object-to-Effect Model, scope (1-1-1-1), trace 6 of 18, Template 6: Not an object of interest, data not sent.

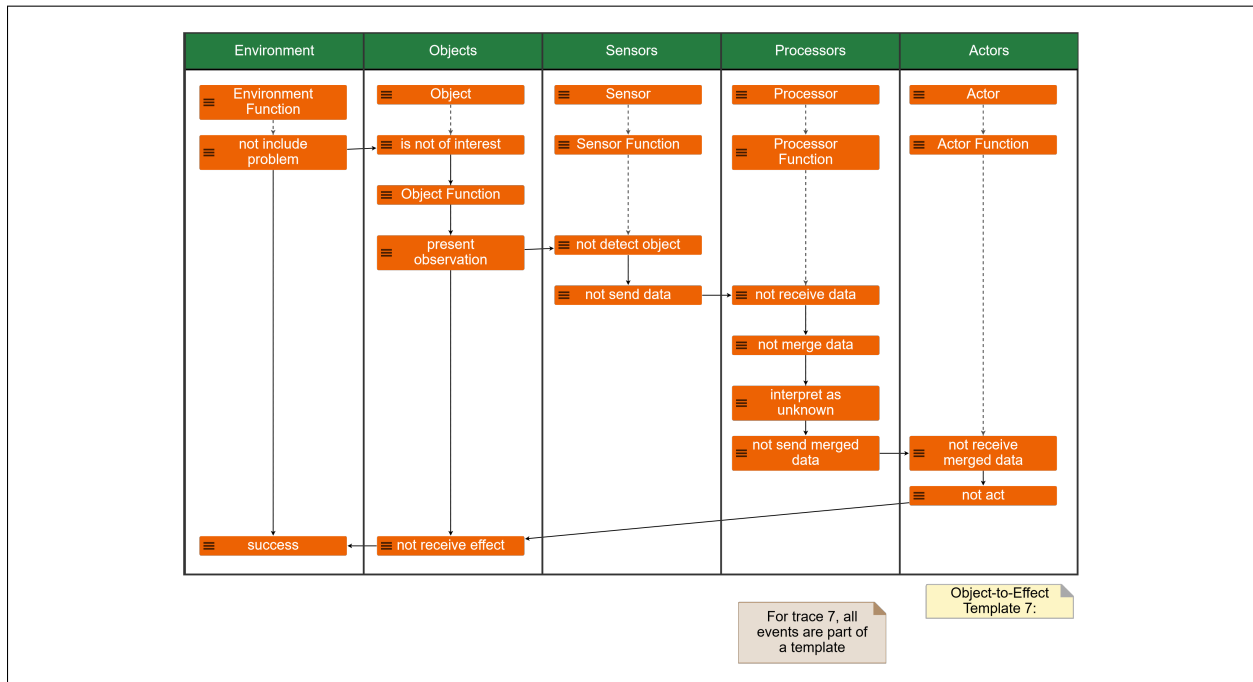


Figure A.19. Object-to-Effect Model, scope (1-1-1-1), trace 7 of 18, Template 7: Not an object of interest, not detected.

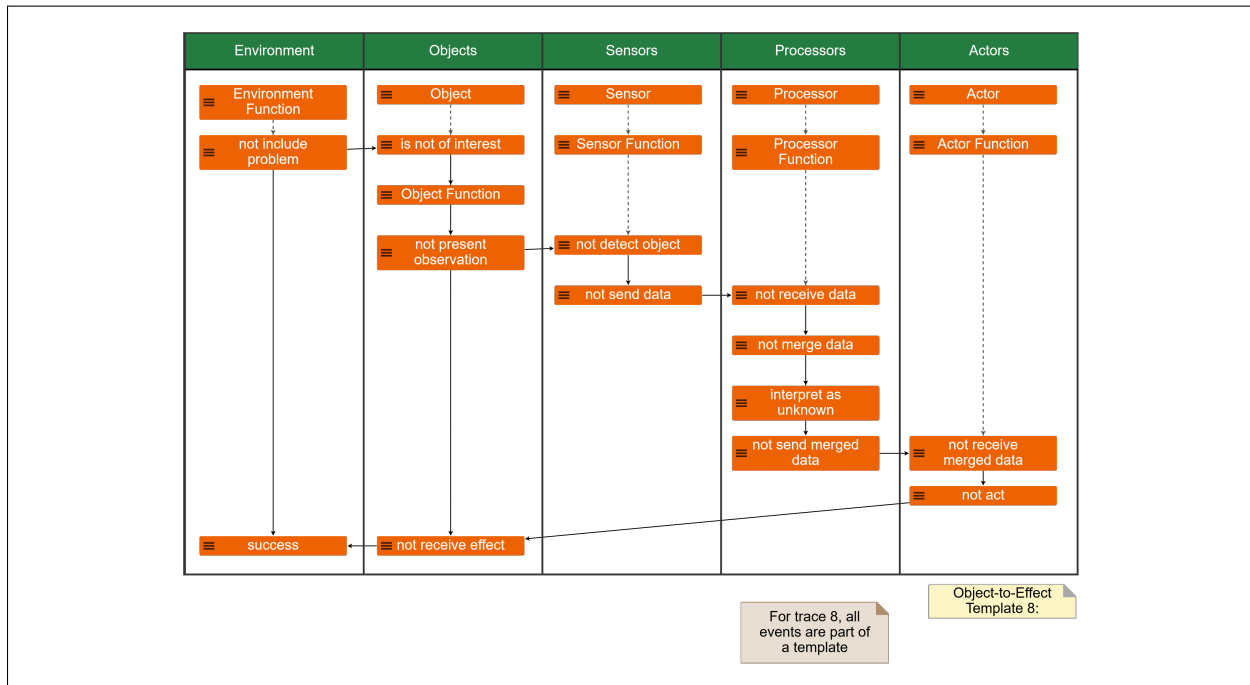


Figure A.20. Object-to-Effect Model, scope (1-1-1-1), trace 8 of 18, Template 8: Not an object of interest, not observable.

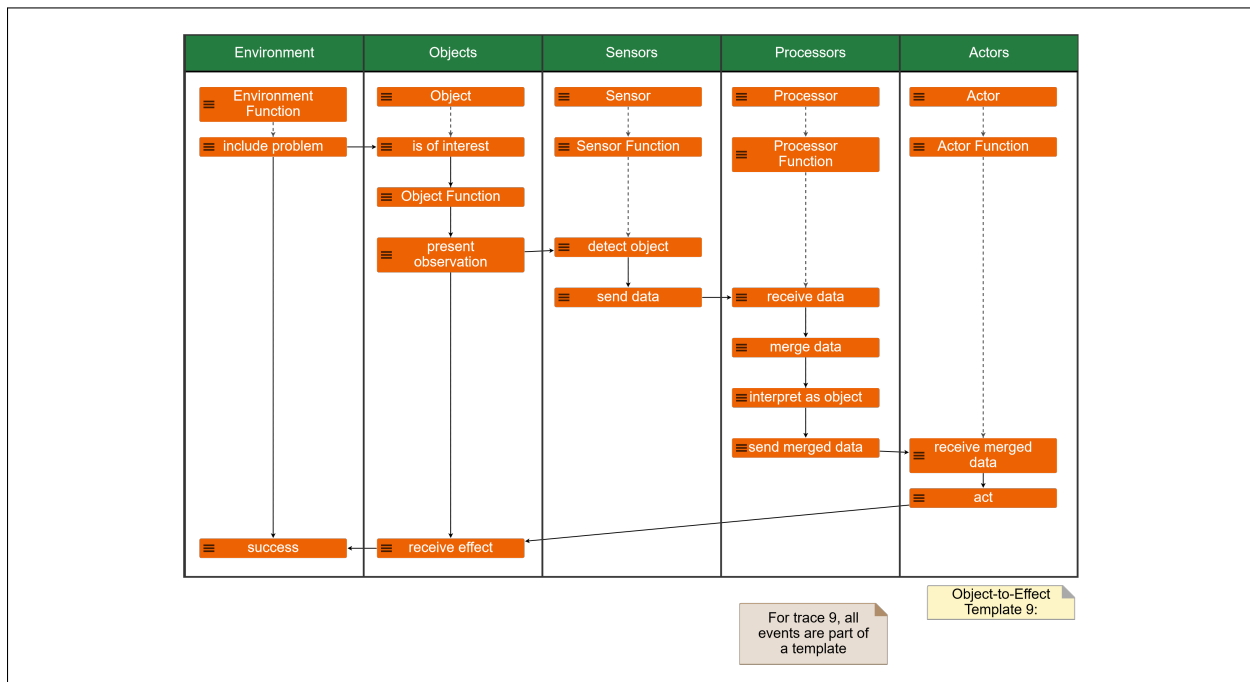


Figure A.21. Object-to-Effect Model, scope (1-1-1-1), trace 9 of 18, Template 9: Object receives effect.

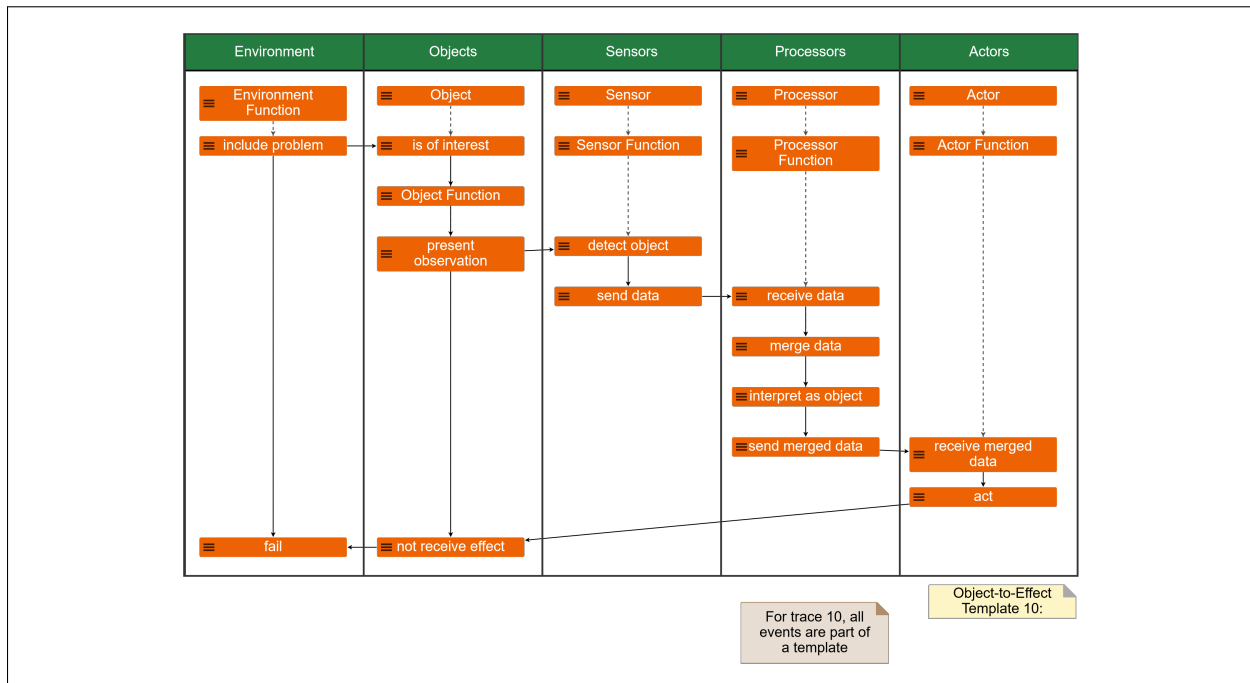


Figure A.22. Object-to-Effect Model, scope (1-1-1-1), trace 10 of 18, Template 10: Object acted upon without effect.

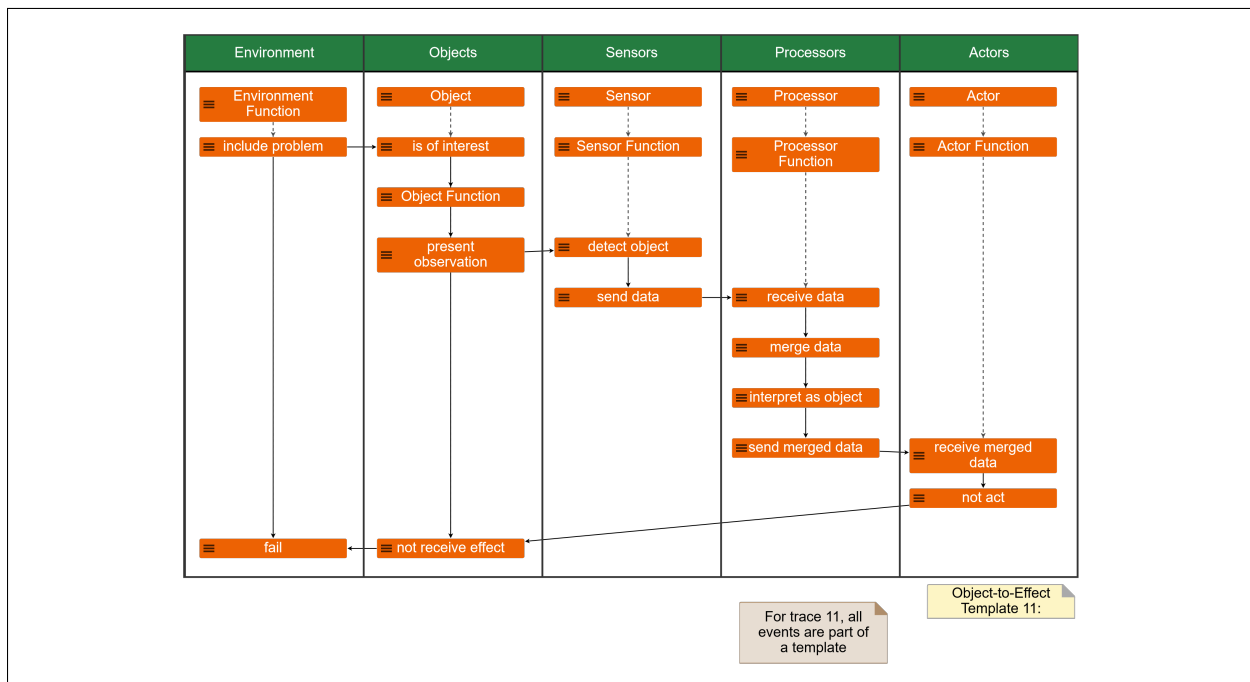


Figure A.23. Object-to-Effect Model, scope (1-1-1-1), trace 11 of 18, Template 11: Object of interest, merged data received, but no action taken.

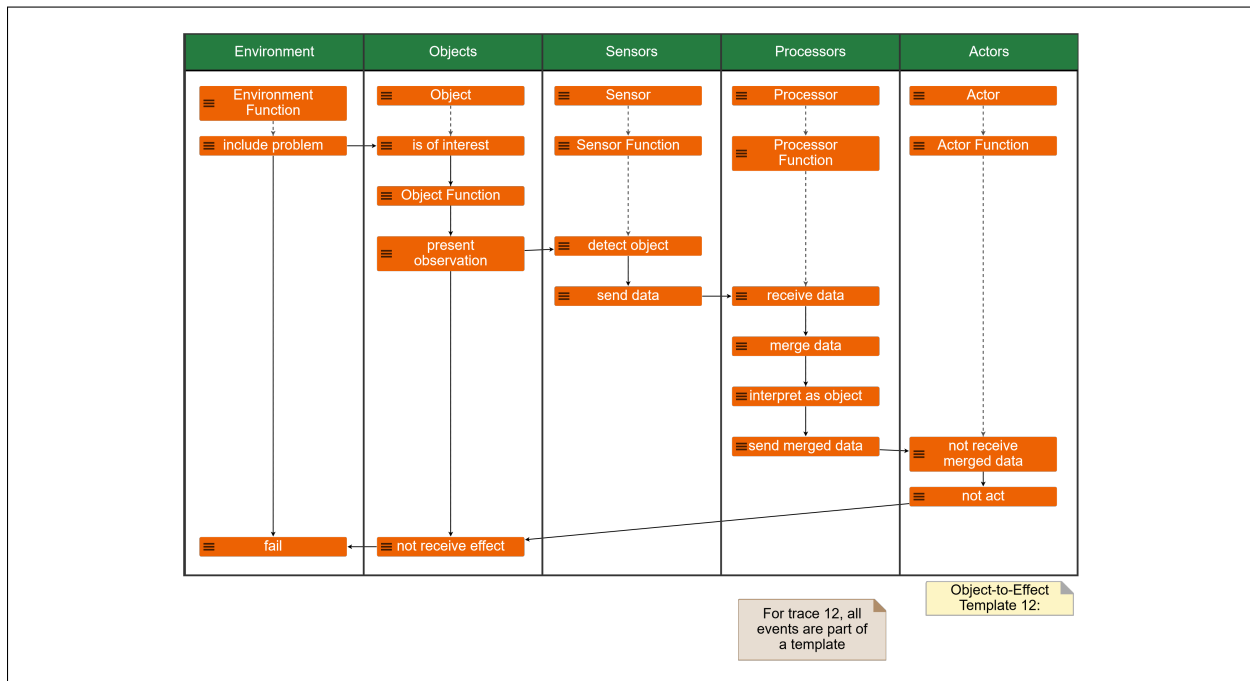


Figure A.24. Object-to-Effect Model, scope (1-1-1-1), trace 12 of 18, Template 12: Object of interest, merged data not received.

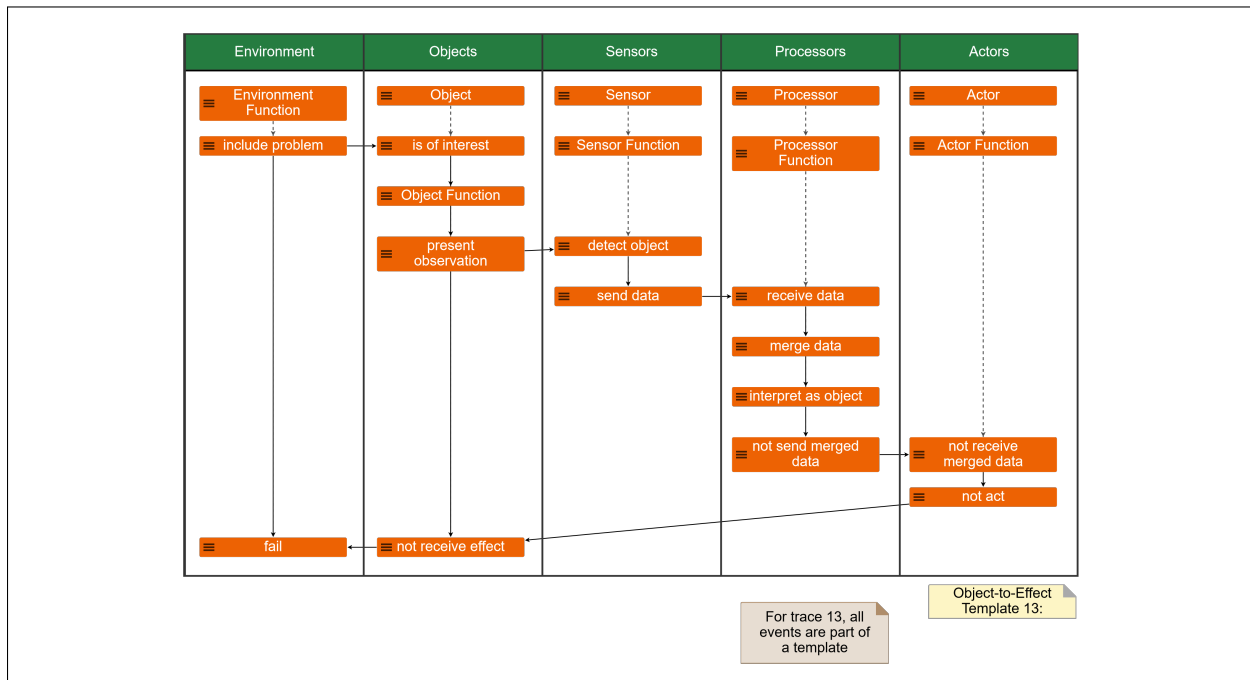


Figure A.25. Object-to-Effect Model, scope (1-1-1-1), trace 13 of 18, Template 13: Object of interest, merged data not sent.

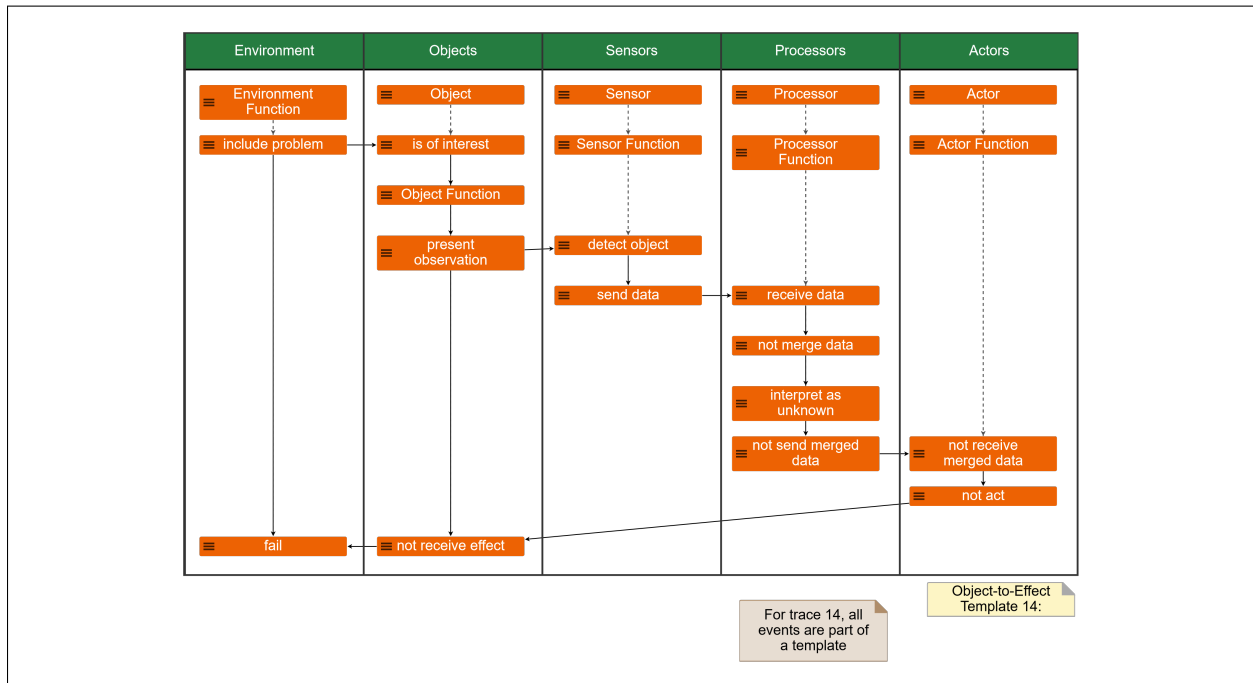


Figure A.26. Object-to-Effect Model, scope (1-1-1-1), trace 14 of 18, Template 14: Object of interest, interpreted as unknown.

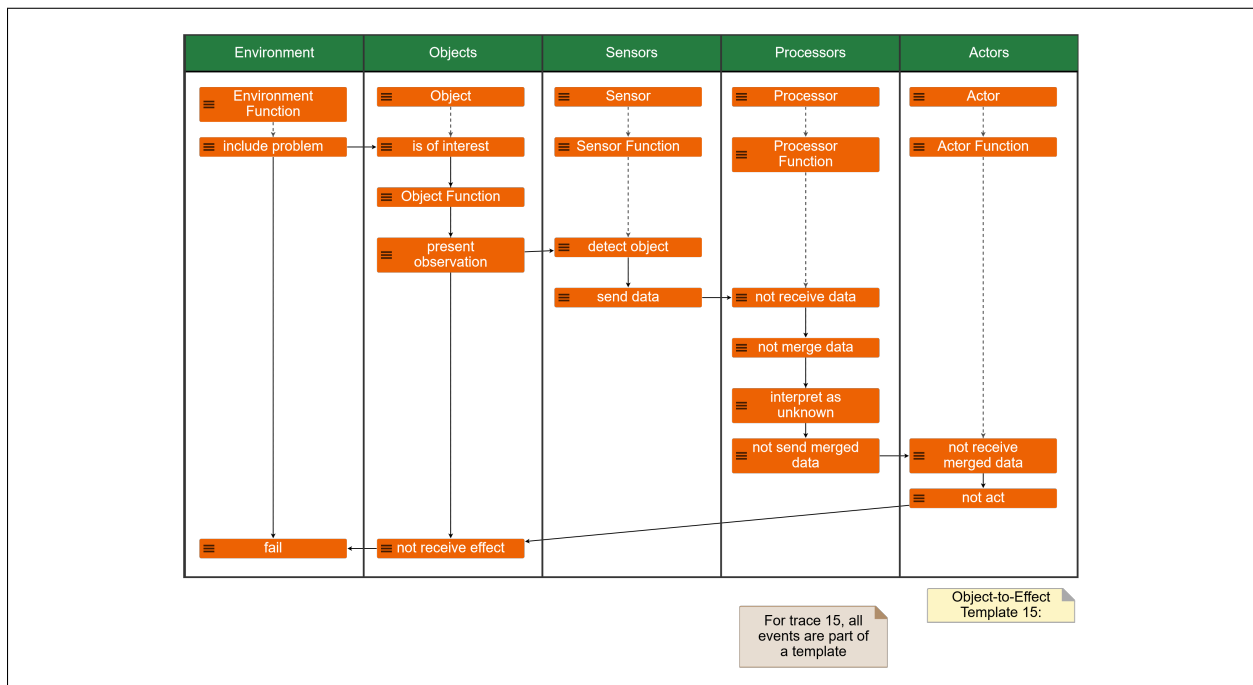


Figure A.27. Object-to-Effect Model, scope (1-1-1-1), trace 15 of 18, Template 15: Object of interest, data not received.

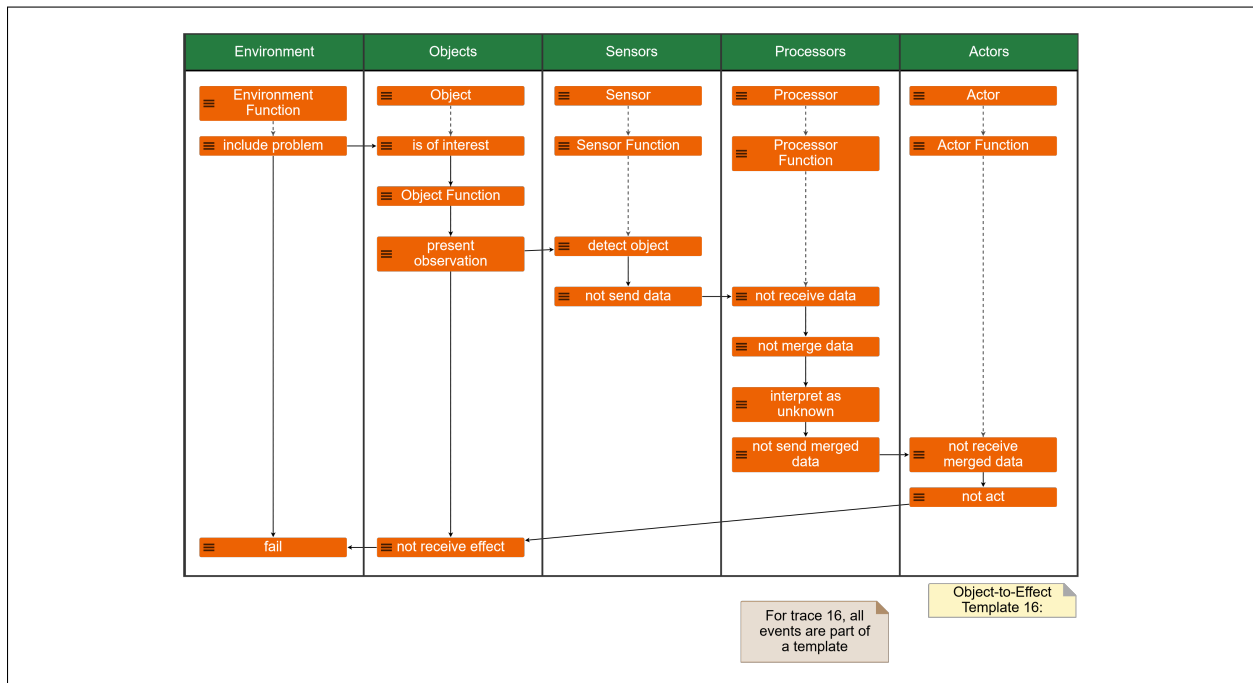


Figure A.28. Object-to-Effect Model, scope (1-1-1-1), trace 16 of 18, Template 16: Object of interest, data not sent.

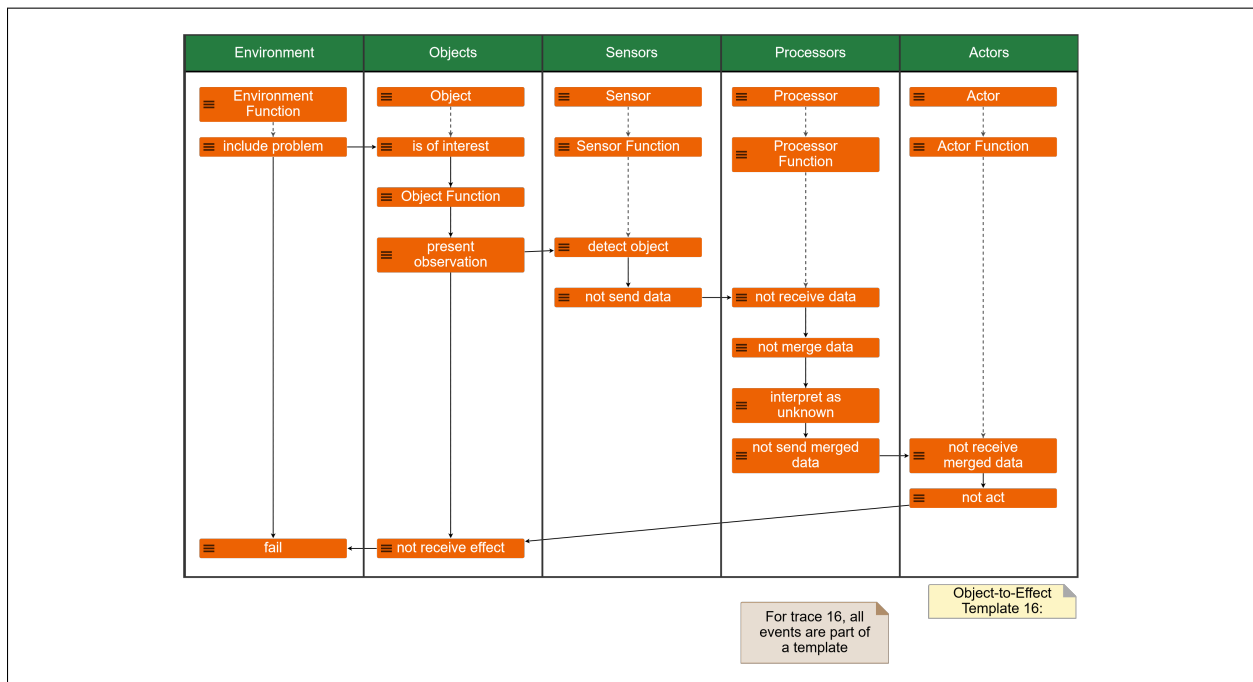


Figure A.29. Object-to-Effect Model, scope (1-1-1-1), trace 17 of 18, Template 17: Object of interest, not detected.

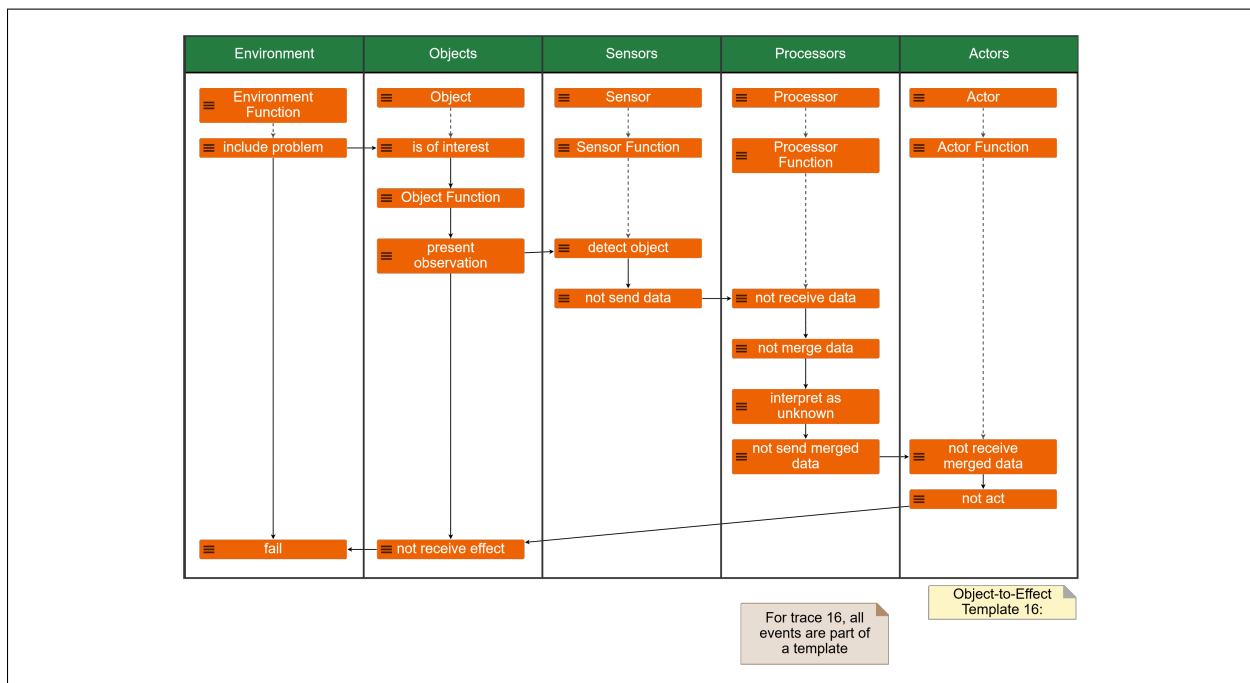


Figure A.30. Object-to-Effect Model, scope (1-1-1-1), trace 18 of 18, Template 18: Object of interest, not observable.

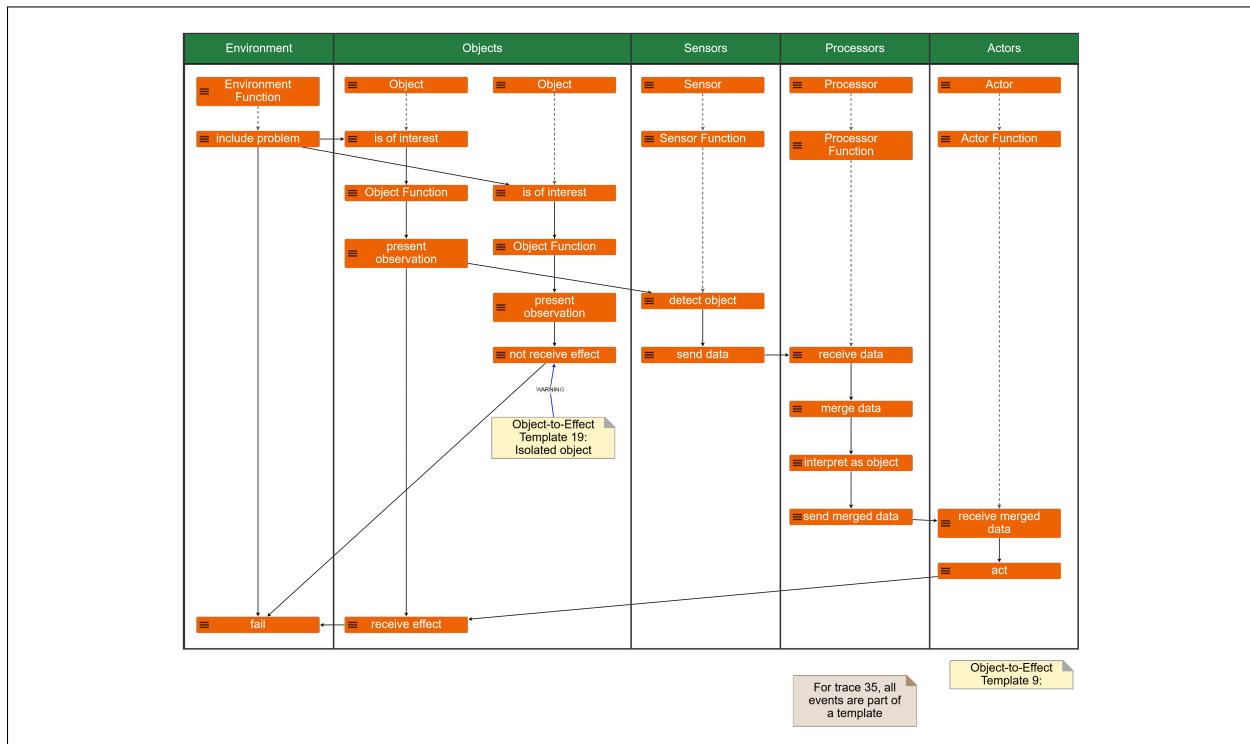


Figure A.31. Object-to-Effect Model, scope (2-1-1-1), trace 35 of 72, Templates 9 and 19: Isolated Object.

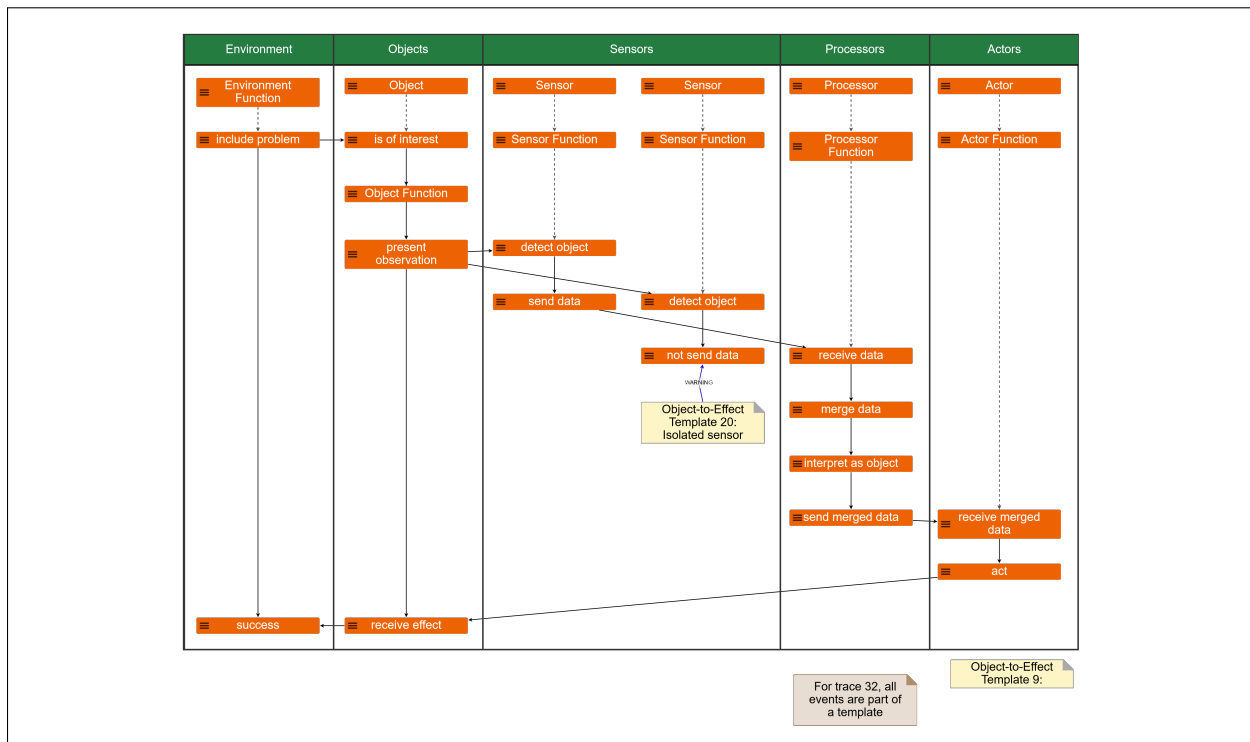


Figure A.32. Object-to-Effect Model, scope (1-2-1-1), trace 32 of 70, Templates 9 and 20: Isolated Sensor.

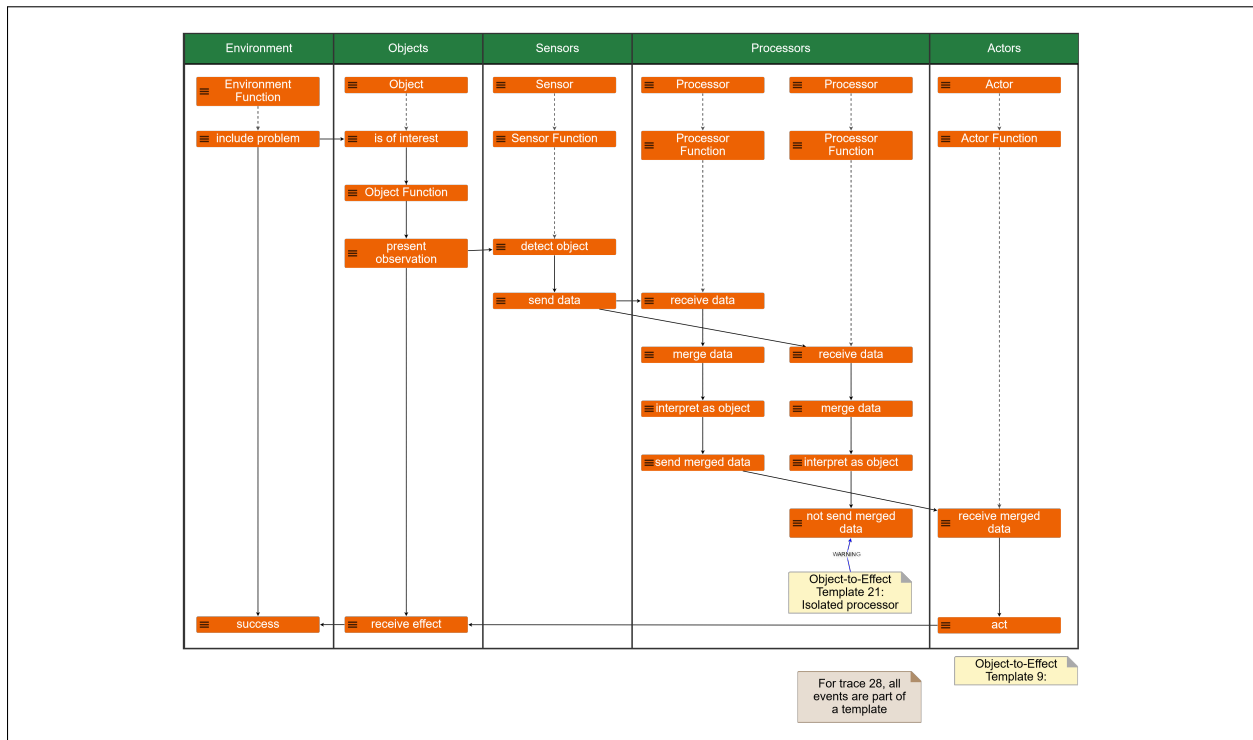


Figure A.33. Object-to-Effect Model, scope (1-1-2-1), trace 28 of 66, Templates 9 and 21: Isolated Processor.

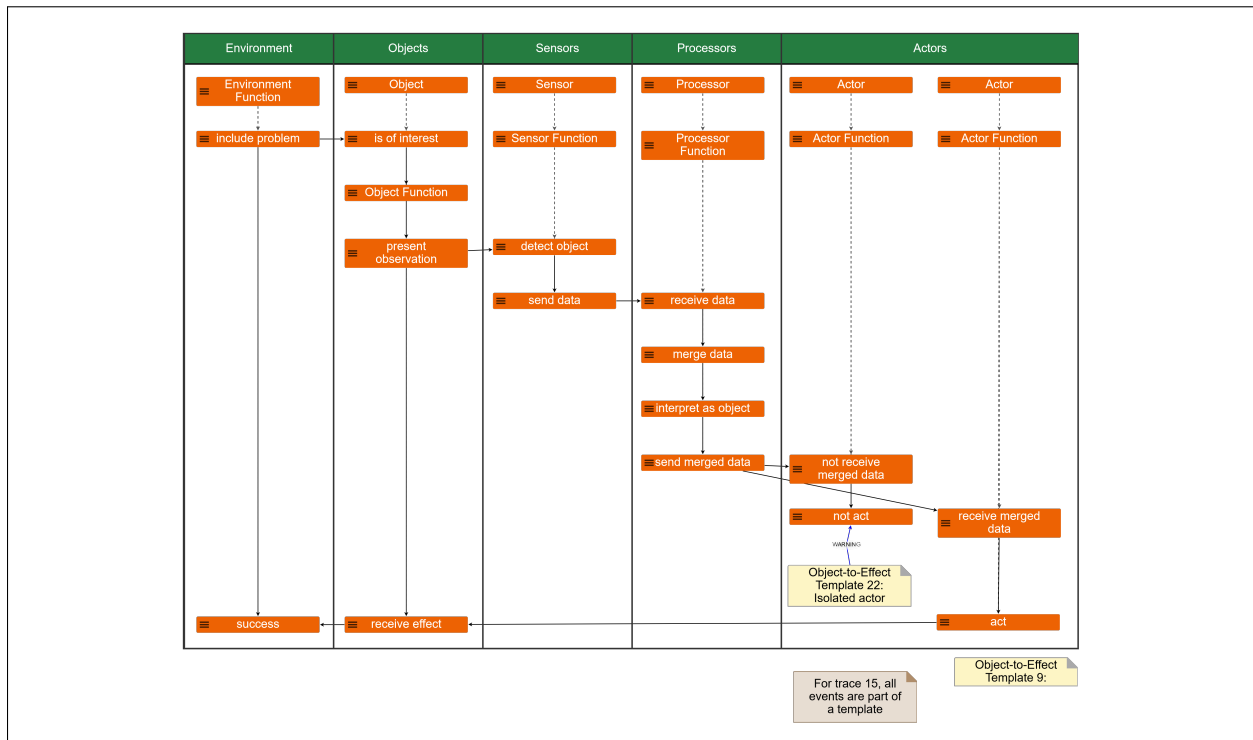


Figure A.34. Object-to-Effect Model, scope (1-1-1-2), trace 15 of 30, Templates 9 and 22: Isolated Actor.

A.2.2 Object-to-Effect Model: MP Code Listing

```
SCHEMA    objectToEffectv22
/* Derived from MITRE paper
   by Dahmann et al. (2017)
   and Antul et al. (2018)

John Quartuccio
First derived 30 Mar 2018
Updated 07 APR 2018
Updated 14 JUL 2018 v3
Updated 15 AUG 2018 v4
Updated 13 OCT 2018 v5
Updated 16 OCT 2018 v6
Updated 30 OCT 2018 v7
Updated 27 JUN 2019 v8
Updated 20 JUL 2019 v9
Updated 31 JUL 2019 v10
Updated 31 Aug 2019 v11
Updated 15 Sep 2019 v12
Updated 20 Sep 2019 v13
Updated 24 Sep 2019 v14
Updated 02 Oct 2019 v15
Updated 12 Oct 2019 v16
Updated 14 Oct 2019 v17
Updated 15 Oct 2019 v18
Updated 19 Oct 2019 v19
Updated 20 Oct 2019 v20, for publication
Updated 29 Dec 2019 v21, alternate structure
Updated 29 Dec 2019 v22, for final dissertation

/*****
/*****Attributes Associated with each Event*****/
/*****
/*
/* Attribute definitions indicated as Boolean expressions      */
/* These attributes serve as meta-data for the template query */
ATTRIBUTES{bool  isEnvironmentFunction,
               isObjectFunction,
               isSensorFunction,
               isProcessorFunction,
               isActorFunction,
               favorable, unfavorable,
               Ex, nEx,                /*existence of problem*/
               It, nIt,                /*object of interest */
               Pr, nPr,                /*present observation */
               Se, Ob, Or, De, Ac,     /*OODA */
               nSe, nOb, nOr, nDe, nAc, /*broken OODA */
               Tx, Rx, Re, Int,        /*transmit, receive, */
               nTx, nRx, nRe, nInt,    /*relay, interrupt */
               Eff, nEff,              /*effect on Object */
               fl, scs;                /*outcome */
};

/* Attributes used to identify whether events are in a template*/
ATTRIBUTES{bool  inTemplate; /*indication of event in template */
           number count,    /*number of events not in template*/
           tracecount; /*number of traces with events not*/
                      /*in any template*/
};
```

```

/*****
/*****ROOT, FUNCTIONS, AND INTERACTIONS*****/
/****Environment*****/
ROOT Environment: { Environment_Function }
  BUILD{inTemplate:=true;};
Environment_Function:
  ( not_include_problem      | include_problem )
  ( success                  | fail )
  BUILD{inTemplate:=true; isEnvironmentFunction:=true;};

/*Environment Attributes*/
not_include_problem :
  BUILD{nEx:= true;    favorable:=true; inTemplate:=true;};
include_problem :
  BUILD{ Ex:= true; unfavorable:=true; inTemplate:=true;};

success :
  BUILD{scs:= true;    favorable:=true; inTemplate:=true;};
fail :
  BUILD{fl := true; unfavorable:= true;inTemplate:=true;};

/****Objects*****/
ROOT Objects:      { + <1> Object      + }
  BUILD{inTemplate:=true;};
Object:      { ( is_not_of_interest | is_of_interest      )
               Object_Function }
  BUILD{inTemplate:=true;};
Object_Function:
  ( present_observation | not_present_observation )
  ( receive_effect      | not_receive_effect      )
  BUILD{inTemplate:=true; isObjectFunction:=true; };

COORDINATE $a:( is_not_of_interest | is_of_interest ),
             $b:( present_observation | not_present_observation)
DO ADD $a PRECEDES $b; OD;

/*Object Attributes*/
is_not_of_interest :
  BUILD{nIt:= true;    favorable:= true;};
is_of_interest :
  BUILD{ It:= true; unfavorable:= true;};

present_observation :
  BUILD{ Pr:= true;    favorable:= true;};
not_present_observation :
  BUILD{nPr:= true; unfavorable:= true;};

receive_effect :
  BUILD{ Eff:= true;};
not_receive_effect :
  BUILD{nEff:= true;};

```

```

/* Interaction #1: Problem state precedes Object identity      */
/**/
COORDINATE      $p: (not_include_problem | include_problem)
DO COORDINATE $t: (is_of_interest      | is_not_of_interest)
DO ADD $p PRECEDES $t; OD; OD;

/* Interaction #2: The effect precedes outcome state          */
/**/
COORDINATE      $h : (receive_effect      | not_receive_effect)
DO COORDINATE $s : (success | fail)
DO ADD $h PRECEDES $s; OD; OD;

/*Constraint #1. DEFINITION: Problem state                    */
/**/
ENSURE (      #include_problem == 1 -> #is_of_interest >= 1);
ENSURE (#not_include_problem == 1 -> #is_of_interest == 0);

/*Constraint #2. DEFINITION: Success and failure              */
/**/
IF (#fail == 1) THEN
    ENSURE EXISTS $x: Object_Function
        ((#is_of_interest BEFORE $x -
          #not_receive_effect IN $x == 0)
        OR
         (#is_not_of_interest BEFORE $x -
          #receive_effect IN $x == 0));
FI;

IF (#success == 1) THEN
    ENSURE FOREACH $x:Object_Function
        (#is_of_interest BEFORE $x -
         #receive_effect IN $x == 0);

    ENSURE FOREACH $x:Object_Function
        (#is_not_of_interest BEFORE $x -
         #not_receive_effect IN $x == 0);
FI;

```

```

/****Sensors*****/
/**/
ROOT Sensors:      { + <l> Sensor      + }
    BUILD{inTemplate:=true;
        COORDINATE $x:Sensor DO $x.inTemplate := true; OD;};

Sensor: Sensor_Function;
Sensor_Function:
    ( detect_object | not_detect_object )
    ( send_data      | not_send_data      )
    BUILD{inTemplate:=true; isSensorFunction:=true;};

/*Sensor Attributes*/
detect_object :
    BUILD{ Ob:= true; Se:= true;   favorable:= true;};
not_detect_object :
    BUILD{ nOb:= true; nSe:= true; unfavorable:= true;};

send_data :
    BUILD{ Tx:= true;   favorable:= true;};
not_send_data :
    BUILD{ nTx:= true; unfavorable:= true;};

/* Interaction #3: Presentation precedes detection          */
/* Each sensor has one sample of an object                  */
/**/
IF (#Object_Function == #Sensor_Function) THEN

    COORDINATE $p:(present_observation | not_present_observation),
        $d:(detect_object | not_detect_object)

    DO ADD $p PRECEDES $d; OD;

ELSE

COORDINATE      $p:(present_observation | not_present_observation)
DO COORDINATE $d:(detect_object | not_detect_object)

    DO IF ( #(present_observation | not_present_observation)
        BEFORE $d == 0 )
        THEN
        ADD $p PRECEDES $d; FI; OD; OD;
FI;

/*Constraint #3. LOGICAL:  If no observable presentation,    */
/*                          then there is no detection         */
/**/ ENSURE FOREACH $x: detect_object
    (#present_observation BEFORE $x > 0);

/*Constraint #4. LOGICAL:  If there is no detection, then no  */
/*                          data sent                           */
/**/ ENSURE FOREACH $x: send_data (#detect_object BEFORE $x > 0);

```

```

/****Processors*****/
/**/
ROOT Processors: { + <l> Processor + }
    BUILD{inTemplate:=true;
        COORDINATE $x:Processor DO $x.inTemplate := true; OD;};

Processor: Processor_Function;
Processor_Function:
    ( receive_data          | not_receive_data      )
    ( merge_data            | not_merge_data        )
    ( interpret_as_not_object | interpret_as_object    )
                                | interpret_as_unknown )
    ( send_merged_data      | not_send_merged_data   )
    BUILD{inTemplate:=true; isProcessorFunction:=true;};

/*Processor Attributes*/
receive_data :
    BUILD{ Rx:= true;   favorable:= true;};
not_receive_data :
    BUILD{nRx:= true; unfavorable:= true;};

merge_data :
    BUILD{ Or:= true;   favorable:= true;};
not_merge_data :
    BUILD{nOr:= true; unfavorable:= true;};

interpret_as_not_object :
    BUILD{ De:= true; };
interpret_as_object :
    BUILD{ De:= true; };
interpret_as_unknown :
    BUILD{nDe:= true; unfavorable:= true;};

send_merged_data :
    BUILD{ Tx:= true;   favorable:= true;};
not_send_merged_data :
    BUILD{nTx:= true; unfavorable:= true;};

```

```

/* Interaction #4: Send Sensor data precedes its reception */
/* Sensors may send multiple sets of data on the same object, */
/* but each processor operates on only one object */
/**/
IF (#Sensor_Function == #Processor_Function) THEN

    COORDINATE      $Tx:(send_data      | not_send_data),
                   $Rx:(receive_data    | not_receive_data)
    DO ADD $Tx PRECEDES $Rx; OD;
ELSE
COORDINATE      $f: Object_Function
DO COORDINATE $p:(present_observation | not_present_observation)
    SUCH THAT $p IN $f

DO COORDINATE $Tx: send_data
DO COORDINATE $Rx:(receive_data | not_receive_data)
DO IF ( ( #(send_data | not_send_data) BEFORE $Rx == 0 ) OR
        ($p BEFORE $Tx AND $p BEFORE $Rx) ) THEN
    ADD $Tx PRECEDES $Rx; FI; OD; OD; OD; OD;

COORDINATE      $f: Object_Function
DO COORDINATE $p:(present_observation | not_present_observation)
    SUCH THAT $p IN $f

DO COORDINATE $nTx: not_send_data
DO COORDINATE $nRx: not_receive_data
DO IF ( ( #(send_data | not_send_data) BEFORE $nRx == 0 ) OR
        ($p BEFORE $nTx AND $p BEFORE $nRx) ) THEN
    ADD $nTx PRECEDES $nRx; FI; OD; OD; OD; OD;
FI;

/*Constraint #5. LOGICAL: If no data sent, then no data received*/
/**/ ENSURE FOREACH $x: receive_data  (#send_data BEFORE $x > 0);

/*Constraint #6. LOGICAL: If no data is received, then cannot */
/* merge data */
/**/ ENSURE FOREACH $x: merge_data (#receive_data BEFORE $x > 0);

/*Constraint #7. LOGICAL: If there is no merged data, then */
/* is no interpretation */
/**/ ENSURE FOREACH $x: not_merge_data
    (#interpret_as_unknown AFTER $x == 1);
/**/ ENSURE FOREACH $x: merge_data
    (#interpret_as_unknown AFTER $x == 0);

/*Constraint #8. LOGICAL: If no merged data, then cannot */
/* send merged data */
/**/ ENSURE FOREACH $x: not_merge_data
    (#send_merged_data AFTER $x == 0);

/*Constraint #9. DESIGN REQUIREMENT: If received data on a */
/* friendly, then don't interpret as hostile */
/**/ ENSURE FOREACH $x: is_not_of_interest, $y: receive_data
    ($x BEFORE $y -> #interpret_as_object AFTER $x == 0);

/*Constraint #10. DESIGN REQUIREMENT: If received data on a */
/* hostile, then don't interpret as friendly */
/**/ ENSURE FOREACH $x: is_of_interest, $y: receive_data
    ($x BEFORE $y -> #interpret_as_not_object AFTER $x == 0);

/*Constraint #x. SIMPLIFICATION: If there is merged data, then*/
/* always send the data */
/* ENSURE FOREACH $x: merge_data
    (#send_merged_data AFTER $x > 0);

```



```

/****Actors*****/
/**/
ROOT Actors:      { + <l> Actor      + }
    BUILD{inTemplate:=true;
        COORDINATE $x:Actor DO $x.inTemplate:=true; OD;};

Actor: Actor_Function;
Actor_Function:
    ( receive_merged_data      | not_receive_merged_data)
    ( act                      | not_act                )
    BUILD{inTemplate:=true; isActorFunction:=true;};

/*Actor Attributes*/
receive_merged_data :
    BUILD{ Rx:= true;   favorable:= true;};
not_receive_merged_data :
    BUILD{nRx:= true; unfavorable:= true;};

act :
    BUILD{ Ac:= true; };
not_act :
    BUILD{nAc:= true; };

/*Conditional Actor Attributes*/
/**/
COORDINATE $x: (is_of_interest | is_not_of_interest)
DO COORDINATE $y: (act | not_act)
DO
    IF ($x IS is_of_interest AND $y IS act AND $x BEFORE $y)
    THEN $y.favorable := true; FI;

    IF ($x IS is_of_interest AND $y IS not_act AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;

    IF ($x IS is_not_of_interest AND $y IS act AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;

    IF ($x IS is_not_of_interest AND $y IS not_act
        AND $x BEFORE $y)
    THEN $y.favorable := true; FI;

OD; OD;

/**/
COORDINATE $x: ( interpret_as_unknown
                | interpret_as_object
                | interpret_as_not_object)
DO COORDINATE $y: (act | not_act)
DO
    IF ($x IS interpret_as_unknown AND $y IS not_act
        AND $x BEFORE $y)
    THEN $y.favorable := true; FI;

    IF ($x IS interpret_as_unknown AND $y IS act
        AND $x BEFORE $y)
    THEN $y.unfavorable := true; FI;

OD; OD;

```

```

/* Interaction #5: Send track precedes reception */
/**/
IF (#Processor_Function == #Actor_Function) THEN

COORDINATE $Tx:(send_merged_data | not_send_merged_data ),
           $Rx:(receive_merged_data | not_receive_merged_data)
DO ADD $Tx PRECEDES $Rx; OD;

ELSE

COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
                SUCH THAT $p IN $f
DO COORDINATE $Tx:(send_merged_data)
DO COORDINATE $Rx:(receive_merged_data | not_receive_merged_data)
DO
  IF ( ( #(send_merged_data | not_send_merged_data )
          BEFORE $Rx == 0 ) OR
        ($p BEFORE $Tx AND $p BEFORE $Rx) ) THEN
    ADD $Tx PRECEDES $Rx; FI; OD; OD; OD; OD;

COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
                SUCH THAT $p IN $f
DO COORDINATE $nTx: not_send_merged_data
DO COORDINATE $nRx: not_receive_merged_data
DO
  IF ( ( #(send_merged_data | not_send_merged_data )
          BEFORE $nRx == 0 ) OR
        ($p BEFORE $nTx AND $p BEFORE $nRx) ) THEN
    ADD $nTx PRECEDES $nRx; FI; OD; OD; OD; OD;

FI;

/* Interaction #6: Action precedes effect */
/**/
COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
                SUCH THAT $p IN $f
DO COORDINATE $s: act
DO COORDINATE $h: (receive_effect | not_receive_effect)
DO IF ($p BEFORE $s AND $p BEFORE $h) THEN
  ADD $s PRECEDES $h; FI; OD; OD; OD; OD;

COORDINATE $f: Object_Function
DO COORDINATE $p: (present_observation | not_present_observation)
                SUCH THAT $p IN $f
DO COORDINATE $s: not_act
DO COORDINATE $h: not_receive_effect
DO IF ($p BEFORE $s AND $p BEFORE $h) THEN
  ADD $s PRECEDES $h; FI; OD; OD; OD; OD;

/*Constraint #11. LOGICAL: If the merged data is not sent, then*/
/* merged data not received */
/**/ ENSURE FOREACH $x:receive_merged_data
      (#send_merged_data BEFORE $x > 0);

```

```

/*Constraint #12. LOGICAL: If there is no action, then the */
/*                      object does not receive an effect */
/**/ ENSURE FOREACH $x:receive_effect      (#act BEFORE $x > 0);

/*Constraint #13. DESIGN REQUIREMENT: If no data received, then*/
/*                      do not act */
/**/ ENSURE FOREACH $x:act      (#receive_merged_data BEFORE $x > 0);

/*Constraint #14. DESIGN REQUIREMENT: If interpreted as a */
/*                      friendly, or unknown, then don't act */
/**/ ENSURE FOREACH $x:interpret_as_not_object(#act AFTER $x==0);
/**/ ENSURE FOREACH $x:interpret_as_unknown    (#act AFTER $x==0);

/*Constraint #15. DESIGN REQUIREMENT: Each action has no more */
/*                      than one effect */
/**/ ENSURE FOREACH $x:act      (#receive_effect AFTER $x <= 1);

/*Constraint #16. LOGICAL: If not present observation then not */
/*                      receive an effect */
/**/ ENSURE FOREACH $x:Object_Function
      (#receive_effect IN $x +
       #not_present_observation IN $x <= 1);

/*****END OF ROOT, FUNCTIONS, ATTRIBUTES AND INTERACTIONS*****/

```

```

/*****
/*****Template Library for the Object-to-Effect Model*****/
/*****
/* This template library employs the attribute definitions from*/
/* the repository, and may be employed directly, without the */
/* natural language from any domain area. */
/*****

/* Sense To Act Template 1: Not an Object of interest */
/**/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
                                     $identity BEFORE $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
AND $detect IN $sf
AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
AND $detect IN $sf
AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
AND $receive IN $pf
AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
AND $merge IN $pf
AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
AND $decide IN $pf
AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
AND $sendMrg IN $pf
AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.Rx
AND $receiveMrg IN $af
AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
AND $act IN $af
AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
AND $effect IN $of
AND $act PRECEDES $effect
AND $presentObs PRECEDES $effect)

```

```

DO SAY("Object-to-Effect Template 1:");

$identity.inTemplate      := true;
$presentObs.inTemplate    := true;
$detect.inTemplate        := true;
$transmit.inTemplate      := true;
$receive.inTemplate       := true;
$merge.inTemplate         := true;
$decide.inTemplate        := true;
$sendMrg.inTemplate       := true;
$receiveMrg.inTemplate     := true;
$sact.inTemplate          := true;
$effect.inTemplate        := true;

OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

/*****/

```

```

/*****
/* Template 2: Not an Object of interest, track not received */
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 2:");

$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 3: Not an Object of interest, track not sent *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 3:");

$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 4:Not an Object of interest, interpreted as unknown*/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 4:");

$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```



```

/*****
/* Template 5: Not an Object of interest, data not received */
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 5:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 6: Not an Object of interest, data not sent *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.nTx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 6:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 7: Not an Object of interest, not detected *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.nSe
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.nTx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 7:");

$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 8: Not an Object of interest, not observable *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.nIt)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs:  $$EVENT SUCH THAT ($presentObs.nPr
    AND $presentObs IN $of)

DO COORDINATE $detect:      $$EVENT SUCH THAT ($detect.nSe
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit:    $$EVENT SUCH THAT ($transmit.nTx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive:     $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge:       $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide:      $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg:     $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg:  $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act:         $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect:      $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 8:");
$identity.inTemplate      := true;
$presentObs.inTemplate    := true;
$detect.inTemplate        := true;
$transmit.inTemplate      := true;
$receive.inTemplate       := true;
$merge.inTemplate         := true;
$decide.inTemplate        := true;
$sendMrg.inTemplate       := true;
$receiveMrg.inTemplate    := true;
$act.inTemplate           := true;
$effect.inTemplate        := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 9: Object of interest receives effect *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.Rx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.Ac
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.Eff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 9:"); MARK;
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 10: Object of interest acted upon with no effect */
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.Rx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.Ac
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 10:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 11: Object of interest, no action *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.Rx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 11:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 12: Object of interest, track not received *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.Tx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 12:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```



```

/*****
/* Template 13: Object of interest, track not sent *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.Or
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.De
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 13:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 14: Object of interest interpreted as unknown *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 14:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 15: Object of interest, data not received *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.Se
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 15:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 16:  Object of interest, data not sent *****/
**/
COORDINATE $identity:  $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of:  $$EVENT SUCH THAT ($of.isObjectFunction AND
                                         $identity PRECEDES $of)
DO COORDINATE $sf:  $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf:  $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af:  $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs:  $$EVENT SUCH THAT ($presentObs.Pr
AND $presentObs IN $of)

DO COORDINATE $detect:  $$EVENT SUCH THAT ($detect.Se
AND $detect IN $sf
AND $presentObs BEFORE $detect)

DO COORDINATE $transmit:  $$EVENT SUCH THAT ($transmit.nTx
AND $detect IN $sf
AND $detect BEFORE $transmit)

DO COORDINATE $receive:  $$EVENT SUCH THAT ($receive.nRx
AND $receive IN $pf
AND $transmit BEFORE $receive)

DO COORDINATE $merge:  $$EVENT SUCH THAT ($merge.nOr
AND $merge IN $pf
AND $receive BEFORE $merge)

DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.nDe
AND $decide IN $pf
AND $merge BEFORE $decide)

DO COORDINATE $sendMrg:  $$EVENT SUCH THAT ($sendMrg.nTx
AND $sendMrg IN $pf
AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg:  $$EVENT SUCH THAT ($receiveMrg.nRx
AND $receiveMrg IN $af
AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act:  $$EVENT SUCH THAT ($act.nAc
AND $act IN $af
AND $receiveMrg BEFORE $act)

DO COORDINATE $effect:  $$EVENT SUCH THAT ($effect.nEff
AND $effect IN $of
AND $act PRECEDES $effect
AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 16:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 17: Object of interest not detected *****/
**/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
                                         $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.Pr
AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.nSe
AND $detect IN $sf
AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.nTx
AND $detect IN $sf
AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
AND $receive IN $pf
AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
AND $merge IN $pf
AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
AND $decide IN $pf
AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
AND $sendMrg IN $pf
AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
AND $receiveMrg IN $af
AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
AND $act IN $af
AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
AND $effect IN $of
AND $act PRECEDES $effect
AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 17:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 18: Object of interest not observable *****/
COORDINATE $identity: $$EVENT SUCH THAT ($identity.It)
DO COORDINATE $of: $$EVENT SUCH THAT ($of.isObjectFunction AND
    $identity PRECEDES $of)
DO COORDINATE $sf: $$EVENT SUCH THAT ($sf.isSensorFunction)
DO COORDINATE $pf: $$EVENT SUCH THAT ($pf.isProcessorFunction)
DO COORDINATE $af: $$EVENT SUCH THAT ($af.isActorFunction)

DO COORDINATE $presentObs: $$EVENT SUCH THAT ($presentObs.nPr
    AND $presentObs IN $of)

DO COORDINATE $detect: $$EVENT SUCH THAT ($detect.nSe
    AND $detect IN $sf
    AND $presentObs BEFORE $detect)

DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.nTx
    AND $detect IN $sf
    AND $detect BEFORE $transmit)

DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.nRx
    AND $receive IN $pf
    AND $transmit BEFORE $receive)

DO COORDINATE $merge: $$EVENT SUCH THAT ($merge.nOr
    AND $merge IN $pf
    AND $receive BEFORE $merge)

DO COORDINATE $decide: $$EVENT SUCH THAT ($decide.nDe
    AND $decide IN $pf
    AND $merge BEFORE $decide)

DO COORDINATE $sendMrg: $$EVENT SUCH THAT ($sendMrg.nTx
    AND $sendMrg IN $pf
    AND $decide BEFORE $sendMrg)

DO COORDINATE $receiveMrg: $$EVENT SUCH THAT ($receiveMrg.nRx
    AND $receiveMrg IN $af
    AND $sendMrg BEFORE $receiveMrg)

DO COORDINATE $act: $$EVENT SUCH THAT ($act.nAc
    AND $act IN $af
    AND $receiveMrg BEFORE $act)

DO COORDINATE $effect: $$EVENT SUCH THAT ($effect.nEff
    AND $effect IN $of
    AND $act PRECEDES $effect
    AND $presentObs PRECEDES $effect)

DO SAY("Object-to-Effect Template 18:");
$identity.inTemplate := true;
$presentObs.inTemplate := true;
$detect.inTemplate := true;
$transmit.inTemplate := true;
$receive.inTemplate := true;
$merge.inTemplate := true;
$decide.inTemplate := true;
$sendMrg.inTemplate := true;
$receiveMrg.inTemplate := true;
$act.inTemplate := true;
$effect.inTemplate := true;
OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD; OD;

```

```

/*****
/* Template 19: Isolated object *****/

COORDINATE $is2:  $$EVENT SUCH THAT
                    ( ($is2.It OR $is2.nIt ) )

DO COORDINATE $of2:  $$EVENT SUCH THAT ( $of2.isObjectFunction
                    AND $is2 BEFORE $of2 )

DO COORDINATE $show2: $$EVENT SUCH THAT
                    ( $show2 IN $of2
                    AND ( $show2.Pr OR $show2.nPr )
                    )

DO COORDINATE $effect2: $$EVENT SUCH THAT
                    ( $effect2 IN $of2
                    AND ( $effect2.Eff OR $effect2.nEff )
                    AND $show2 PRECEDES $effect2 )

DO

    IF (#($$EVENT) PRECEDES $effect2 == 1) THEN

        ADD SAY("Object-to-Effect Template 19: Isolated object")
            WARNING $effect2;

        $is2.inTemplate      := true;
        $show2.inTemplate    := true;
        $effect2.inTemplate  := true;
        FI;

OD; OD; OD; OD; /* OD;

/*****
/* Template 20: Isolated sensor *****/

COORDINATE $sensor:  $$EVENT SUCH THAT
                    $sensor.isSensorFunction

DO COORDINATE $detect:  $$EVENT SUCH THAT
                    ( $detect IN $sensor
                    AND ($detect.Se OR $detect.nSe ) )

DO COORDINATE $transmit:  $$EVENT SUCH THAT
                    ( $transmit IN $sensor
                    AND ($transmit.Tx OR $transmit.nTx )
                    AND $detect PRECEDES $transmit )

DO COORDINATE $outcome:  $$EVENT SUCH THAT
                    ( ( $outcome.scs OR $outcome.fl )
                    AND NOT $outcome AFTER $transmit )

DO
ADD SAY("Object-to-Effect Template 20: Isolated sensor")
    WARNING $transmit;

    $detect.inTemplate      := true;
    $transmit.inTemplate    := true;

OD; OD; OD; OD;

```

```

/*****
/* Template 21: Isolated Processor *****/
COORDINATE $ops2:      $$EVENT SUCH THAT
                        $ops2.isProcessorFunction

DO COORDINATE $receive2:  $$EVENT SUCH THAT
                        ( $receive2 IN $ops2
                          AND ($receive2.Rx OR $receive2.nRx) )

DO COORDINATE $fuse2:    $$EVENT SUCH THAT
                        ( $fuse2 IN $ops2
                          AND ($fuse2.Or OR $fuse2.nOr )
                          AND $receive2 PRECEDES $fuse2 )

DO COORDINATE $interpret2:  $$EVENT SUCH THAT
                        ( $interpret2 IN $ops2
                          AND ( $interpret2.De OR $interpret2.nDe )
                          AND $fuse2 PRECEDES $interpret2 )

DO COORDINATE $transmit2:  $$EVENT SUCH THAT
                        ( $transmit2 IN $ops2
                          AND ($transmit2.Tx OR $transmit2.nTx)
                          AND $interpret2 PRECEDES $transmit2 )

DO COORDINATE $outcome:   $$EVENT SUCH THAT
                        ( ( $outcome.scs OR $outcome.fl )
                          AND NOT $outcome AFTER $transmit2 )

DO
ADD SAY("Object-to-Effect Template 21: Isolated processor")
                        WARNING $transmit2;

    $receive2.inTemplate := true;
    $fuse2.inTemplate    := true;
    $interpret2.inTemplate := true;
    $transmit2.inTemplate := true;

    OD; OD; OD; OD; OD; OD;

/*****
/* Template 22: Isolated actor *****/
COORDINATE $actor:      $$EVENT SUCH THAT
                        $actor.isActorFunction

DO COORDINATE $receive:  $$EVENT SUCH THAT
                        ( $receive IN $actor
                          AND ($receive.Rx OR $receive.nRx ) )

DO COORDINATE $act:      $$EVENT SUCH THAT
                        ( $act IN $actor
                          AND ($act.Ac OR $act.nAc )
                          AND $receive PRECEDES $act )

DO COORDINATE $outcome:  $$EVENT SUCH THAT
                        ( ( $outcome.scs OR $outcome.fl )
                          AND NOT $outcome AFTER $act )

DO
ADD SAY("Object-to-Effect Template 22: Isolated actor")
                        WARNING $act;

    $receive.inTemplate := true;
    $act.inTemplate      := true;

    OD; OD; OD; OD;

```



```

/*****
/*****General Template Library *****/
/*****
/* This template library employs the attribute definitions from*/
/* the repository, and may be employed directly, without the */
/* natural language from any domain area. */
/*
/* These are general templates, applicable to all models that */
/* employ semantics that are consistent within the defined */
/* ontology. */
/*
/*****
IF (0) THEN /* Set to one in Order to Implement Segments *****/
/*****
/*****Identifying Favorable and Unfavorable Events*****/
/*****

/**/
COORDINATE $parent: $$EVENT
DO COORDINATE $positiveEvents: $$EVENT
    SUCH THAT $positiveEvents.favorable
DO
    IF ($positiveEvents IN $parent) THEN
        SAY ("Positive event: " $parent " does " $positiveEvents);
    FI; OD; OD;

/**/
COORDINATE $parent: $$EVENT
DO COORDINATE $negativeEvents: $$EVENT
    SUCH THAT $negativeEvents.unfavorable
DO
    IF ($negativeEvents IN $parent) THEN
        SAY ("Negative event: " $parent " does " $negativeEvents);
    FI; OD; OD;

/* Identify the Send-Receive */
/**/
COORDINATE $transmit: $$EVENT SUCH THAT $transmit.Tx
DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
    AND $transmit PRECEDES $receive)
DO
    SAY("Transmit-Receive: Tx:" $transmit " , Rx:" $receive );
    OD; OD;

/* Identify a Relay, such that receive and transmit are in the */
/* same event */
/**/
COORDINATE $receive: $$EVENT SUCH THAT $receive.Rx
DO COORDINATE $transmit: $$EVENT SUCH THAT $transmit.Tx
DO
    IF ($transmit.Rx) THEN
        SAY("Relay found: both Rx and Tx in the same event "
            $transmit );
    FI; OD; OD;

```

```

/* Identify orientation that depends upon reception */
/**/
COORDINATE $receive:    $$EVENT SUCH THAT $receive.Rx
DO COORDINATE $orient:  $$EVENT SUCH THAT $orient.Or
DO
    IF ($orient.Rx) THEN
        SAY("Orientation and reception in the same event: "
            $orient );
    FI; OD; OD;

/* Identify the OODA LOOP */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $orient:  $$EVENT SUCH THAT ($orient.Or
                        AND $orient AFTER $observe)
DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.De
                        AND $decide AFTER $orient)
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
                        AND $act AFTER $decide)
DO
    SAY("OODA loop: " $observe ", " $orient ", " $decide ", "
        $act);
OD; OD; OD; OD;

/* Identify a Coordinated OODA LOOP */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.Ob
DO COORDINATE $transmit: $$EVENT SUCH THAT ($transmit.Tx
                        AND $transmit AFTER $observe)
DO COORDINATE $receive: $$EVENT SUCH THAT ($receive.Rx
                        AND $receive AFTER $transmit)
DO COORDINATE $orient:  $$EVENT SUCH THAT ($orient.Or
                        AND $orient AFTER $observe)
DO COORDINATE $decide:  $$EVENT SUCH THAT ($decide.De
                        AND $decide AFTER $orient)
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
                        AND $act AFTER $decide)
DO MARK;
SAY("Coordinated OODA loop: " $observe ", " $transmit ", "
    $receive ", " $orient ", "
    $decide ", " $act);
OD; OD; OD; OD; OD; OD;

/* Identify an uninformed negative action */
/**/
COORDINATE $observe:    $$EVENT SUCH THAT $observe.nOb
DO COORDINATE $act:     $$EVENT SUCH THAT ($act.Ac
                        AND $act.unfavorable)
DO
    SAY("Uninformed Action: " $observe ", then " $act); MARK;
OD; OD;

/**/
FI; /* End of General Template Segment of Schema *****/

```

```

/*****
/*****GENERAL STATISTICS*****/
/*****
/*Mark the traces that have events not in a template*****/
**/
COORDINATE      $a:$SEVENT SUCH THAT NOT $a.inTemplate
  DO ADD SAY("not_in_template") not_in_template $a;
    GLOBAL.count += 1;
    count += 1;
    MARK;
  OD;
/*****
/* Statistic Reports for each trace *****/
**/
REPORT GeneralReport {};
IF (count > 0) THEN
  SAY("For trace " trace_id ", " count
    " event(s) not part of a template")
=>GeneralReport;
ELSE
  SAY("For trace " trace_id
    ", all events are part of a template")
=>GeneralReport;
FI;

IF(count > 0) THEN GLOBAL.tracecount += 1; FI;

SHOW  GeneralReport;
CLEAR GeneralReport;

/*****
/* Global Reports for the entire schema *****/
**/
GLOBAL
IF (GLOBAL.count > 0 ) THEN
  SAY("Summary: " GLOBAL.count " event(s) in " GLOBAL.tracecount
    " trace(s) are not part of a template, as marked")
    => GeneralReport;
ELSE
  SAY("Summary: All events in all traces are part of a template")
    => GeneralReport;
FI;
SHOW GeneralReport;

/*End of General Report Section*/

```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:

Probability Calculations

This appendix outlines a MATLAB[®] script developed in order to derive the probability of each trace for a constrained Monterey Phoenix (MP) behavior model. The script recreates the derivation of all potential traces, applies the constraints reducing the number of traces to only those that are valid, then applies a prorating scheme that is consistent with the trimmed branches of the underlying belief network.

1. Populate the probability belief network without constraints.
2. Add conditional constraints that are consistent with the model, such that $p = 0$ for alternatives that satisfy the constraint.
3. Prorate the remaining structure of the belief network such that the total probability of each sample space is equal to one. The “TrimBranch.m” script is the key to successful implementation of the algorithm.

Applying these principles to a simple example, the Decision Model, and the Object-To-Effect Model were conducted. The following sections list these results and the associated scripts, as published within MATLAB[®].

B.1 Probability Calculation of the Example Problem

This script produces an example of a constrained probability network

Contents

- Script: constraintExample.m
- clear the command window, clear memory, close all files
- Assign then number of alternatives, e, to each node
- Initialize the probability matrix, Pn to an even distribution
- Assign initial probabilities and constraints to the belief network
- Print the probability matrix as modified by the constraints
- Prorate the matrix and print these final results using the trimBranch function

Called Functions

- trimBranch

Script: constraintExample.m

Written by John Quartuccio

```
% 09 May 2017 initial baseline
% 10 Aug 2019 updated for multiple sets of alternatives
% 11 Nov 2019 finalized for publication
```

clear the command window, clear memory, close all files

```
clc;clear all;close all;
```

Assign then number of alternatives, e, to each node

```
e = [2,3,2];
%
```

Initialize the probability matrix, Pn to an even distribution

```
P = zeros(12,1); Pn = ones(12,3)/2; Pn(:,2) = 1/3; Pt = prod(Pn,2);
r = 0; traces = 0;
```

Assign initial probabilities and constraints to the belief network

written as the probability equal to zero if the condition applies

```
r=0;
for i = 1:2 % a:
    for j = 1:3 % b:
        for k = 1:2 % c:
            r = r +1;
```

```

% First column
if (i == 1)
    Pn(r,1)=.4;
end
if (i == 2)
    Pn(r,1)=.6;
end

% Second column
if (j == 1)
    Pn(r,2)=.1;
end
if (j == 2)
    Pn(r,2)=.3;
end
if (j == 3)
    Pn(r,2)=.6;
end

% Third column
if (i == 1 && j == 1)
    if(k == 1) Pn(r,3)=0; end
    if(k == 2) Pn(r,3)=0; end
end

if (i == 1 && j == 2)
    if(k == 1) Pn(r,3)=0.45; end
    if(k == 2) Pn(r,3)=0.55; end
end

if (i == 1 && j == 3)
    if(k == 1) Pn(r,3)=0.6; end
    if(k == 2) Pn(r,3)=0.4; end
end

if (i == 2 && j == 1)
    if(k == 1) Pn(r,3)=.65; end
    if(k == 2) Pn(r,3)=.35; end
end

if (i == 2 && j == 2)
    if(k == 1) Pn(r,3)=.7; end
    if(k == 2) Pn(r,3)=.3; end
end

if (i == 2 && j == 3)
    if(k == 1) Pn(r,3)=.75; end
    if(k == 2) Pn(r,3)=.25; end
end

end
end
end
Pt = prod(Pn,2);

```

Print the probability matrix as modified by the constraints

```

fprintf('Probability matrix, Pn, modified by constraints:\n');
r = 0; traces = 0; %P = zeros(8,1);
for i = 1:2 % a:
    for j = 1:3 % b:
        for k = 1:2 % c:
            r = r + 1;
            traces=traces+1;
            fprintf('%3d %8.5f ',r,Pt(r));
            fprintf('Pa(i=%d)=%5.3f, ',i,Pn(r,1));
            fprintf('Pb(j=%d)=%5.3f, ',j,Pn(r,2));
            fprintf('Pc(k=%d)=%5.3f, ',k,Pn(r,3));
            fprintf('\n');
        end
    end
end
fprintf('total probability: %8.5f\n\n',sum(Pt))

```

Probability matrix, Pn, modified by constraints:

1	0.00000	Pa(i=1)=0.400,	Pb(j=1)=0.100,	Pc(k=1)=0.000,
2	0.00000	Pa(i=1)=0.400,	Pb(j=1)=0.100,	Pc(k=2)=0.000,
3	0.05400	Pa(i=1)=0.400,	Pb(j=2)=0.300,	Pc(k=1)=0.450,
4	0.06600	Pa(i=1)=0.400,	Pb(j=2)=0.300,	Pc(k=2)=0.550,
5	0.14400	Pa(i=1)=0.400,	Pb(j=3)=0.600,	Pc(k=1)=0.600,
6	0.09600	Pa(i=1)=0.400,	Pb(j=3)=0.600,	Pc(k=2)=0.400,
7	0.03900	Pa(i=2)=0.600,	Pb(j=1)=0.100,	Pc(k=1)=0.650,
8	0.02100	Pa(i=2)=0.600,	Pb(j=1)=0.100,	Pc(k=2)=0.350,
9	0.12600	Pa(i=2)=0.600,	Pb(j=2)=0.300,	Pc(k=1)=0.700,
10	0.05400	Pa(i=2)=0.600,	Pb(j=2)=0.300,	Pc(k=2)=0.300,
11	0.27000	Pa(i=2)=0.600,	Pb(j=3)=0.600,	Pc(k=1)=0.750,
12	0.09000	Pa(i=2)=0.600,	Pb(j=3)=0.600,	Pc(k=2)=0.250,

total probability: 0.96000

Prorate the matrix and print these final results using the trimBranch function

```

Pn = trimBranch(Pn,e);
Pt = prod(Pn,2);
fprintf('Probability matrix, Pn, adjusted by trimBranch:\n');
r = 0; traces = 0;
for i = 1:2 % a:
    for j = 1:3 % b:
        for k = 1:2 % c:
            r = r + 1;
            traces=traces+1;
            fprintf('%3d %8.5f ',r,Pt(r));
            fprintf('Pa(i=%d)=%5.3f, ',i,Pn(r,1));
            fprintf('Pb(j=%d)=%5.3f, ',j,Pn(r,2));
            fprintf('Pc(k=%d)=%5.3f, ',k,Pn(r,3));
            fprintf('\n');
        end
    end
end
fprintf('total probability: %8.5f\n\n',sum(Pt))

```

Probability matrix, Pn, adjusted by trimBranch:

1	0.00000	Pa(i=1)=0.400,	Pb(j=1)=0.000,	Pc(k=1)=0.000,


```

2  0.00000  Pa(i=1)=0.400, Pb(j=1)=0.000, Pc(k=2)=0.000,
3  0.06300  Pa(i=1)=0.400, Pb(j=2)=0.350, Pc(k=1)=0.450,
4  0.07700  Pa(i=1)=0.400, Pb(j=2)=0.350, Pc(k=2)=0.550,
5  0.15600  Pa(i=1)=0.400, Pb(j=3)=0.650, Pc(k=1)=0.600,
6  0.10400  Pa(i=1)=0.400, Pb(j=3)=0.650, Pc(k=2)=0.400,
7  0.03900  Pa(i=2)=0.600, Pb(j=1)=0.100, Pc(k=1)=0.650,
8  0.02100  Pa(i=2)=0.600, Pb(j=1)=0.100, Pc(k=2)=0.350,
9  0.12600  Pa(i=2)=0.600, Pb(j=2)=0.300, Pc(k=1)=0.700,
10 0.05400  Pa(i=2)=0.600, Pb(j=2)=0.300, Pc(k=2)=0.300,
11 0.27000  Pa(i=2)=0.600, Pb(j=3)=0.600, Pc(k=1)=0.750,
12 0.09000  Pa(i=2)=0.600, Pb(j=3)=0.600, Pc(k=2)=0.250,
total probability:  1.00000

```

Published with MATLAB® R2014b

Contents

- [Function \[p\] = trimBranchAction\(P,eventnum\)](#)
- [Determine the number of rows and columns based on the matrix, P](#)
- [Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.](#)

```
function [p] = trimBranchAction(P,eventnum)
```

Function [p] = trimBranchAction(P,eventnum)

This function operates on a probability matrix, with each decision based on any number of alternatives defined by the vector eventnum. The algorithm eliminates probabilities that have been constrained from occurring (p=0), prorates the probability of the remaining branches by adjusting the remaining alternatives, and returns the resulting trimmed probability matrix.

```

% P is the probability matrix
% eventnum: matrix that identifies the number of alternatives of each event

% Written by John Quartuccio
% 21 Apr 2017
% 28 Jul 2019 updated for general numbers of alternatives per event
% 24 Aug 2019 finalized for publication

```

Determine the number of rows and columns based on the matrix, P

```

numrows = length(P); % the number of rows, each representing a trace
numcols = size(P,2); % the number of columns, each representing an event

%% Zero out probabilities for the entire row (trace) if any zero is found
%% in any column (event) of the trace
% for m = 1:numrows % m is index for each row (trace)
%     for n = 1:numcols % n is index for each column (event)
%         if (P(m,n) == 0)
%             P(m,:) = 0;
%         end % end of if
%     end % end of column index, n (event)
% end % end of row index, m (trace)

```

Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.

Initialize values for the step size and number of events per step. The step size indicates the number of alternatives associated with the event

```
step = numrows; % reset the step index
neps = 1; % initialize the number of events per step
%
for n = numcols :-1:1 % index, n is the column number
    step = step/eventnum(n); % derive the number of steps in the event column
    neps = neps*eventnum(n); % derive the number of events per step
    % fprintf('\n\nColumn %d:\n',n) % debug
    % fprintf('    number of steps: %d\n',step) % debug
    % fprintf('    number of events per step: %d\n',neps) % debug
    %
    m = 0; % reset the row index, m
    for s = 1:step % index, s, for the step associated with each set of
        % events, eventnum(n)
        stepsum = 0; % reset the step sum of probabilities
        numnon0 = 0; % reset the number of events with non-zero probability
        num0 = 0; % reset the number of events with zero probability
        % fprintf('    step: %d\n',s) % debug
        for e = 1:neps
            m = m + 1;
            stepsum = stepsum + P(m,n);
            if (P(m,n)==0)
                num0 = num0 + 1;
                row0 = m;
            % fprintf('        zero row index: %d\n',row0) % debug
            end % end if
            if (P(m,n)~=0)
                numnon0 = numnon0 + 1;
            end % end if
        end % end of event block: e = 1:neps

        % Determine the portion of probability to adjust each nonzero value
        Padj = (1 - stepsum*eventnum(n)/neps) * neps / eventnum(n) / numnon0;
        % fprintf('        step sum of p: %5.3f\n',stepsum) % debug
        % fprintf('        number non-zero: %d\n',numnon0) % debug
        % fprintf('        total p in step: %d\n',(1*neps/eventnum(n))) % debug
        % fprintf('        adjusted value: %5.3f\n', Padj) % debug

        % If sum of probabilities is zero, adjust the "upstream" set to zero
        if (stepsum == 0)
            for e = 1:neps
                P((m-e+1),(n-1)) = 0;
            end % end of event block
        end % end if (stepsum == 0)

        % Allocate residual probability to non-zero values
        if (int16(stepsum*1000) ~= int16(1*neps/eventnum(n)*1000) ...
            && stepsum ~=0)
            for e = 1:neps
                if (P((m-e+1),n) ~= 0)
                    P(m-e+1,n) = P(m-e+1,n) + Padj;
                end % end if
            end % end of event block
        end % end of event block
    end % end of event block: n = numcols :-1:1
end % end of column block: n = numcols :-1:1
```

```
        end % end if (stepsum ~= 1)

    end % end of step

end % end of pass

p=P;
```

```
end %end of function
```

Published with MATLAB® R2014b

B.2 Probability Calculation of the Decision Model

This script determines the probability for the occurrence of each trace of the Decision Model

Contents

- [Script: constraintDecision.m](#)
- [Clear memory](#)
- [Set initial probabilities](#)
- [Apply constraints](#)
- [Determine the probability of each trace](#)
- [Apply the prorating algorithm to trim the branches](#)
- [Determine additional properties](#)

Called Functions

- [constraintAppliedDecision](#)
- [constraintPrintDecision](#)
- [constraintProbabilityDecision](#)
- [constraintPropertiesDecision](#)
- [trimBranchAction](#)

Script: constraintDecision.m

applies constraints to a decision network, where each decision point has the possibility of one of two alternatives. The constraints reset some of these probabilities to zero. Using the function trimBranch, the remaining probabilities are prorated in a way that is consistent with the applied constraints.

written by: John Quartuccio

```
% 21 Apr 2017 version 1: basic
% 09 May 2017 version 2: Modified to adjust probabilities
% 21 Oct 2018 version 3: Added comments for clarification
% 09 Nov 2019 finalized for publication
```

Clear memory

```
clc;clear all;close all;
```

Set initial probabilities

```
Pn = ones(256,8)/2;
constraintProbabilityDecision;

% Calculate the probability of each trace based on initial assignments
Pt = prod(Pn,2);
```

```
% Assign then number of alternatives, e, to each node
e = ones(8,1)*2;
```

Apply constraints

```
constraintAppliedDecision;
```

Determine the probability of each trace

```
Pt = prod(Pn,2);

% Print the probability of each trace
fprintf('\nConstrained probability:\n');
constraintPrintDecision;
```

```
Constrained probability:
1, 1, 0.21952, Template 1: No problem
2, 33, 0.09408, Template 5: No problem, incorrect leader
3, 65, 0.05880, Template 3: No problem, incorrect subordinate
4, 97, 0.02520, Template 7: No problem, incorrect perception
5, 129, 0.05488, Template 2: Problem exists, correct perception
6, 161, 0.02352, Template 6: Problem, incorrect leader perception
7, 176, 0.01008, Template 9: Leader fails to consider subordinate
8, 192, 0.00840, Template 10: Problem, subordinate fails to communicate
9, 193, 0.01470, Template 4: Problem exists, incorrect subordinate
10, 225, 0.00630, Template 8: Problem, incorrect perception
11, 240, 0.00270, Template 11: Problem, comm incorrect perception
12, 256, 0.00900, Template 12: Problem, all went wrong

total probability: 0.52718
```

Apply the prorating algorithm to trim the branches

```
Pn = trimBranchAction(Pn,e);

% Re-calculate the probability of each trace
Pt = prod(Pn,2);
%
% Print the probability of each trace
fprintf('\nTrimmed probability:\n');
constraintPrintDecision;
```

```
Trimmed probability:
1, 1, 0.39200, Template 1: No problem
2, 33, 0.16800, Template 5: No problem, incorrect leader
3, 65, 0.16800, Template 3: No problem, incorrect subordinate
4, 97, 0.07200, Template 7: No problem, incorrect perception
5, 129, 0.09800, Template 2: Problem exists, correct perception
6, 161, 0.02352, Template 6: Problem, incorrect leader perception
7, 176, 0.01008, Template 9: Leader fails to consider subordinate
8, 192, 0.00840, Template 10: Problem, subordinate fails to communicate
9, 193, 0.04200, Template 4: Problem exists, incorrect subordinate
10, 225, 0.00630, Template 8: Problem, incorrect perception
```

```

11, 240, 0.00270, Template 11: Problem, comm incorrect perception
12, 256, 0.00900, Template 12: Problem, all went wrong

total probability: 1.00000

```

Determine additional prperties

```
constraintPropertiesDecision;
```

Probability of success: 0.969820

Probability of failure: 0.030180

Published with MATLAB® R2014b

Apply constraints to the Decision Model

Contents

- [Script: constraintAppliedDecision.m](#)

Script: constraintAppliedDecision.m

```

ii = 0; traces = 0;
for i = 1:2 % a: environment condition : column 1
    for j = 1:2 % b: subordinate perception : column 2
        for k = 1:2 % c: leader perception : column 3
            for m = 1:2 % d: communication : column 4
                for n = 1:2 % e: reception : column 5
                    for p = 1:2 % f: decision : column 6
                        for q = 1:2 % g: action : column 7
                            for r = 1:2 % h: outcome : column 8
                                ii = ii + 1;
                                % Constraint 1: if no problem (i=1),
                                % then have a successful outcome (r=1, column 8).
                                if (i == 1)
                                    if (r == 1)
                                        if (Pn(ii,8)~=0) Pn(ii,8) = 1; end
                                    else
                                        Pn(ii,8) = 0;
                                    end
                                end
                                % Constraint 2: if no communication (m=2),
                                % then no reception (n=2, column 5)
                                if (m == 2)
                                    if (n == 2)
                                        if (Pn(ii,5)~=0) Pn(ii,5) = 1; end
                                    else
                                        Pn(ii,5) = 0;
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```

```

end
% Constraint 3: if leader recognizes (k=1) & no input (n=2),
%   then correct decision (p=1, column 6)
if(k==1 && n==2)
    if(p==1)
        if (Pn(ii,6)~=0) Pn(ii,6) = 1; end
    else
        Pn(ii,6)=0;
    end
end

% Constraint 5: if receive input (n=1),
%   then make a correct decision (p=1, column 6 )
%   and corollary (n=2, then p=2))
if (n == 1)
    if(p==1)
        if (Pn(ii,6)~=0) Pn(ii,6) = 1; end
    else
        Pn(ii,6)=0;
    end
end

if (n == 2)
    if(p==2)
        if (Pn(ii,6)~=0) Pn(ii,6) = 1; end
    else
        Pn(ii,6)=0;
    end
end

% Constraint 6: if correct decision (p=1),
%   then a correct action (q=1) and corollary
if (p == 1)
    if (q == 1)
        if (Pn(ii,7)~=0) Pn(ii,7) = 1; end
    else
        Pn(ii,7) = 0;
    end
end

if (p == 2)
    if (q == 2)
        if (Pn(ii,7)~=0) Pn(ii,7) = 1;end
    else
        Pn(ii,7) = 0;
    end
end

% Constraint 7: if correct action (q=1),
%   then a successful (r=1, column 8) and corollary
if (q == 1)
    if (r == 1)
        if (Pn(ii,8)~=0) Pn(ii,8) = 1; end
    else
        Pn(ii,8) = 0;
    end
end

if (q == 2)
    if (r == 2)
        if (Pn(ii,8)~=0) Pn(ii,8) = 1;end
    else
        Pn(ii,8) = 0;
    end
end

```

```

        end
    end

    end
end
end
end
end
end
end
end
end

% Preserve this constrained matrix, P
P = Pn;

```

Published with MATLAB® R2014b

Print the results

Contents

- [Script constraintPrintDecision.m](#)

Script constraintPrintDecision.m

```

ii = 0; traces=0;
for i = 1:2 % a: environment condition : column 1
    for j = 1:2 % b: subordinate perception : column 2
        for k = 1:2 % c: leader perception : column 3
            for m = 1:2 % d: communication : column 4
                for n = 1:2 % e: reception : column 5
                    for p = 1:2 % f: decision : column 6
                        for q = 1:2 % g: action : column 7
                            for r = 1:2 % h: outcome : column 8
                                ii = ii + 1;
                                if( Pt(ii) > 0 )
                                    traces=traces+1;
                                    fprintf(' %3d, %3d, %8.5f, ',traces,ii,Pt(ii));

                                %
                                fprintf(' P(i=%d)=%4.2f, ',i,Pn(ii,1));
                                fprintf(' P(j=%d)=%4.2f, ',j,Pn(ii,2));
                                fprintf(' P(k=%d)=%4.2f, ',k,Pn(ii,3));
                                fprintf(' P(m=%d)=%4.2f, ',m,Pn(ii,4));
                                fprintf(' P(n=%d)=%4.2f, ',n,Pn(ii,5));
                                fprintf(' P(p=%d)=%4.2f, ',p,Pn(ii,6));
                                fprintf(' P(q=%d)=%4.2f, ',q,Pn(ii,7));
                                fprintf(' P(r=%d)=%4.2f, ',r,Pn(ii,8));

                                end
                                % Template 1: No problem perceived by leader and subordinate
                                if(k==1 && j==1 && i==1 && m==1 && n==1 ...
                                    && p==1 && q==1 && r==1)
                                    fprintf(' Template 1: No problem');
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

```



```

end
% Template 2: Problem exists, correct perception
if(k==1 && j==1 && i==2 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 2: Problem exists, correct perception');
end
% Template 3: No problem, incorrect subordinate perception
if(k==1 && j==2 && i==1 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 3: No problem, incorrect subordinate');
end
% Template 4: Problem exists, incorrect subordinate
if(k==1 && j==2 && i==2 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 4: Problem exists, incorrect subordinate');
end
% Template 5: No problem, incorrect leader perception
if(k==2 && j==1 && i==1 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 5: No problem, incorrect leader');
end
% Template 6: Problem, incorrect leader perception
if(k==2 && j==1 && i==2 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 6: Problem, incorrect leader perception');
end
% Template 7: No problem, incorrect perception of both
if(k==2 && j==2 && i==1 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 7: No problem, incorrect perception');
end
% Template 8: Problem, incorrect perception of both
if(k==2 && j==2 && i==2 && m==1 && n==1 ...
    && p==1 && q==1 && r==1)
    fprintf(' Template 8: Problem, incorrect perception');
end
% Template 9: Leader fails to consider subordinate
if(k==2 && j==1 && i==2 && m==1 && n==2 ...
    && p==2 && q==2 && r==2)
    fprintf(' Template 9: Leader fails to consider subordinate');
end
% Template 10: Problem, subordinate fails to communicate
if(k==2 && j==1 && i==2 && m==2 && n==2 ...
    && p==2 && q==2 && r==2)
    fprintf(' Template 10: Problem, subordinate fails to communicate');
end
% Template 11: Problem, incorrect perception with
% communication
if(k==2 && j==2 && i==2 && m==1 && n==2 ...
    && p==2 && q==2 && r==2)
    fprintf(' Template 11: Problem, comm incorrect perception');
end
% Template 12: Problem, all went wrong
if(k==2 && j==2 && i==2 && m==2 && n==2 ...
    && p==2 && q==2 && r==2)
    fprintf(' Template 12: Problem, all went wrong');
end
if (Pt(ii)>0)
    fprintf('\n');
end;
end

```

```

        end
    end
end
end
end
end
end
end
fprintf('\ntotal probability: %8.5f\n\n',sum(Pt))

```

Published with MATLAB® R2014b

Assign Initial Probabilities

Contents

- [Script: constraintProbabilityDecision](#)

Script: constraintProbabilityDecision

```

ii = 0;
for i = 1:2 % a: environment condition : column 1
    for j = 1:2 % b: subordinate perception : column 2
        for k = 1:2 % c: leader perception : column 3
            for m = 1:2 % d: communication : column 4
                for n = 1:2 % e: reception : column 5
                    for p = 1:2 % f: decision : column 6
                        for q = 1:2 % g: action : column 7
                            for r = 1:2 % h: outcome : column 8
                                ii = ii + 1;
                                % A problem exists (k=2) 20% of the time, column 3.
                                if (i == 2)
                                    Pn(ii,1) = .2;
                                else
                                    Pn(ii,1) = .8;
                                end
                                % The leader has correct perception (k=1) 70% of the time,
                                % column 1.
                                if (k == 1)
                                    Pn(ii,3) = .7;
                                else
                                    Pn(ii,3) = .3;
                                end
                                % The subordinate has correct perception (j=1) 70% of the time,
                                % column 2.
                                if (j == 1)
                                    Pn(ii,2) = .7;
                                else
                                    Pn(ii,2) = .3;
                                end
                                % If the subordinate has an accurate perception (j=1),
                                % the subordinate communicates (m=1) 80% of the time, column 4.
                                if (j == 1)
                                    if (m==1)
                                        Pn(ii,4) = .8;
                                    else

```



```

        end
    end
end
end
end
end
end
end
end
% end
% Ptotal = sum(Pt),
fprintf('Probability of success: %f\n\n',Ps);
fprintf('Probability of failure: %f\n\n',Po);

```

Published with MATLAB® R2014b

Contents

- Function [p] = trimBranchAction(P,eventnum)
- Determine the number of rows and columns based on the matrix, P
- Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.

```
function [p] = trimBranchAction(P,eventnum)
```

Function [p] = trimBranchAction(P,eventnum)

This function operates on a probability matrix, with each decision based on any number of alternatives defined by the vector eventnum. The algorithm eliminates probabilities that have been constrained from occurring ($p=0$), prorates the probability of the remaining branches by adjusting the remaining alternatives, and returns the resulting trimmed probability matrix.

```

% P is the probability matrix
% eventnum: matrix that identifies the number of alternatives of each event

% Written by John Quartuccio
% 21 Apr 2017
% 28 Jul 2019 updated for general numbers of alternatives per event
% 24 Aug 2019 finalized for publication

```

Determine the number of rows and columns based on the matrix, P

```

numrows = length(P); % the number of rows, each representing a trace
numcols = size(P,2); % the number of columns, each representing an event

%% Zero out probabilities for the entire row (trace) if any zero is found
%% in any column (event) of the trace
% for m = 1:numrows % m is index for each row (trace)
%     for n = 1:numcols % n is index for each column (event)
%         if (P(m,n) == 0)
%             P(m,:) = 0;
%         end % end of if
%     end % end of column index, n (event)
% end % end of row index, m (trace)

```

Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.

Initialize values for the step size and number of events per step. The step size indicates the number of alternatives associated with the event

```
step = numrows; % reset the step index
neps = 1; % initialize the number of events per step
%
for n = numcols :-1: 1 % index, n is the column number
    step = step/eventnum(n); % derive the number of steps in the event column
    neps = neps*eventnum(n); % derive the number of events per step
    % fprintf('\n\nColumn %d:\n',n) % debug
    % fprintf('    number of steps: %d\n',step) % debug
    % fprintf('    number of events per step: %d\n',neps) % debug
    %
    m = 0; % reset the row index, m
    for s = 1:step % index, s, for the step associated with each set of
        % events, eventnum(n)
        stepsum = 0; % reset the step sum of probabilities
        numnon0 = 0; % reset the number of events with non-zero probability
        num0 = 0; % reset the number of events with zero probability
        % fprintf('    step: %d\n',s) % debug
        for e = 1:neps
            m = m + 1;
            stepsum = stepsum + P(m,n);
            if (P(m,n)==0)
                num0 = num0 + 1;
                row0 = m;
            % fprintf('    zero row index: %d\n',row0) % debug
            end % end if
            if (P(m,n)~=0)
                numnon0 = numnon0 + 1;
            end % end if
        end % end of event block: e = 1:neps

        % Determine the portion of probability to adjust each nonzero value
        Padj = (1 - stepsum*eventnum(n)/neps) * neps / eventnum(n) / numnon0;
        % fprintf('    step sum of p: %5.3f\n',stepsum) % debug
        % fprintf('    number non-zero: %d\n',numnon0) % debug
        % fprintf('    total p in step: %d\n',(1*neps/eventnum(n))) % debug
        % fprintf('    adjusted value: %5.3f\n', Padj) % debug

        % If sum of probabilities is zero, adjust the "upstream" set to zero
        if (stepsum == 0)
            for e = 1:neps
                P((m-e+1),(n-1)) = 0;
            end % end of event block
        end % end if (stepsum == 0)

        % Allocate residual probability to non-zero values
        if (int16(stepsum*1000) ~= int16(1*neps/eventnum(n)*1000) ...
            && stepsum ~=0)
            for e = 1:neps
                if (P((m-e+1),n) ~= 0)
                    P(m-e+1,n) = P(m-e+1,n) + Padj;
                end % end if
            end % end of event block
        end % end if (stepsum ~= 1)
    end % end of column block: n = numcols :-1: 1
end % end of row block: m = numrows
```

```
end % end of step
```

```
end % end of pass
```

```
p=P;
```

```
end %end of function
```

Published with MATLAB® R2014b

B.3 Probability Calculation of the Object-to-Effect Model

This script determines the constrained probabilities for each trace of the Object-to-Effect Model

Contents

- Script: constraintAction.m script
- Clear memory
- Set initial probabilities
- Assign unique probability conditions for a particular problem
- Assign constraints to the model
- Determine the probability of each trace
- Apply the prorating algorithm to the probability matrix
- Determine additional properties

Called Functions

- constraintAppliedAction
- constraintPrintAction
- constraintProbabilityAction
- constraintPropertiesAction
- trimBranch

Script: constraintAction.m script

written by: John Quartuccio 27 Jul 2019

Scripts and functions used include the following:

```
% constrainProbabilityAction: Script that assigns initial conditional
%                             probability values,

% constraintAppliedAction:   Script that applies constraints to the
%                             probability matrix

% constraintPrintAction:     Script that prints out the probability matrix,

% constraintProperties:       Script that calculates additional properties
%                             such as the probability of success,

% trimBranch:               Function that prorates the residual
%                             probabilities of the constrained matrix
```

Clear memory

```
clc;clear all;close all;
%
```

Set initial probabilities

```
Pn = ones((2^12*3),13)/2; Pn(:,8) = 1/3;
% Assign the number of alternatives, e, to each node
e = ones(33,1)*2; e(8)=3;
```

Assign unique probability conditions for a particular problem

```
constraintProbabilityAction;
```

Assign constraints to the model

```
constraintAppliedAction;
```

Determine the probability of each trace

```
Pt = prod(Pn,2);
%
% Print the probability of each trace
fprintf('\nConstrained probability:\n');
constraintPrintAction;
%
```

```
Constrained probability:
Trace; Ref; P(trace); i; P(a); j; P(b); k; P(c); m; P(d); n; P(e); p; P(f); q; P(g); r; P(h); s; P(aa); t; P(ab); u; P(ac); v; P(ad); w;
1; 7; 0.44686; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 2; 1.000; 1; 1.000
2; 15; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 1; 1.000
3; 31; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 1; 1.000
4; 191; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
```

```

5; 383; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
6; 767; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
7; 1535; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
8; 3871; 0.05000; 1; 0.500; 1; 1.000; 2; 0.100; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
9; 9249; 0.39815; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.999; 1; 0.999; 1; 0.999; 1; 0.990; 1; 0.990; 1; 1.000
10; 9252; 0.00402; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.999; 1; 0.999; 1; 0.999; 1; 0.900; 2; 0.010; 2; 1.000
11; 9256; 0.04469; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.999; 1; 0.999; 1; 0.999; 2; 0.100; 2; 1.000; 2; 1.000
12; 9264; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000
13; 9280; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
14; 9408; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
15; 9600; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
16; 9984; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
17; 10752; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
18; 12288; 0.05000; 2; 0.500; 2; 1.000; 2; 0.100; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000

```

total probability: 0.99910

Apply the proring algorithm to the probability matrix

```

Pn = trimBranch(Pn,e);
%
% Re-calculate the probability of each trace
Pt = prod(Pn,2);
%
% Print the probability of each trace
fprintf('\nTrimmed probability:\n');
constraintPrintAction;

```

```

Trimmed probability:
Trace; Ref; P(trace); i; P(a); j; P(b); k; P(c); m; P(d); n; P(e); p; P(f); q; P(g); r; P(h); s; P(aa); t; P(ab); u; P(ac); v; P(ad); w;
1; 7; 0.44731; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 1.000; 1; 0.999; 1; 0.999; 2; 1.000; 2; 1.000; 1; 1.000
2; 15; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 1.000; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000
3; 31; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 1; 1.000; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
4; 191; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
5; 383; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
6; 767; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
7; 1535; 0.00045; 1; 0.500; 1; 1.000; 1; 0.900; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
8; 3871; 0.05000; 1; 0.500; 1; 1.000; 2; 0.100; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 1; 1.000
9; 9249; 0.39855; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 1; 0.999; 1; 0.999; 1; 0.990; 1; 0.990; 1; 1.000
10; 9252; 0.00402; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 1; 0.999; 1; 0.999; 1; 0.990; 2; 0.010; 2; 1.000
11; 9256; 0.04473; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 1; 0.999; 1; 0.999; 2; 0.100; 2; 1.000; 2; 1.000
12; 9264; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000
13; 9280; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 1; 0.999; 2; 1.000; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
14; 9408; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 1; 0.999; 2; 0.001; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
15; 9600; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 1; 0.999; 2; 0.001; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
16; 9984; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 1; 0.999; 2; 0.001; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
17; 10752; 0.00045; 2; 0.500; 2; 1.000; 1; 0.900; 2; 0.001; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000
18; 12288; 0.05000; 2; 0.500; 2; 1.000; 2; 0.100; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 3; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000; 2; 1.000

```

total probability: 1.00000

Determine additional properties

```
constraintPropertiesAction;
```

```

Probability of success: 0.8986
Probability of failure: 0.1014
Probability of ID as not an object of interest: 0.4482
Probability of ID as object of interest      : 0.4482
Probability of ID as unknown                 : 0.1036

```

Published with MATLAB® R2014b

Apply constraints to the Object-to-Effect Model

Contents

- Script: constraintAppliedAction

Script: constraintAppliedAction

```

trace = 0; traces = 0;
for i = 1:2 % e1: problem state :      column 1
    for j = 1:2 % e2: object of interest : column 2
        for k = 1:2 % e3: present observation : column 3
            for m = 1:2 % e4: detected :      column 4
                for n = 1:2 % e5: send data :      column 5

```



```

for p = 1:2 % e6: receive data :      column 6
for q = 1:2 % e7: merge data :      column 7
for r = 1:3 % e8: interpretation :   column 8
for s = 1:2 % e9: send merged data : column 9
for t = 1:2 % e10: receive merged data: column 10
for u = 1:2 % e11: action :          column 11
for v = 1:2 % e12: effect :          column 12
for w = 1:2 % e13: outcome :         column 13
    trace = trace + 1;

% Constraint 1: Problem related to nature of object of interest
% If no problem exists (i=1),
% then object is not of interest (j=1).
if (i == 1)
    if (j == 1)
        if (Pn(trace,2)~=0) Pn(trace,2) = 1; end
    else
        Pn(trace,2) = 0;
    end
end
% And the corollary if problem (i=2),
% then object is of interest (j=2)
if (i == 2)
    if (j == 2)
        if (Pn(trace,2)~=0) Pn(trace,2) = 1; end
    else
        Pn(trace,2) = 0;
    end
end

% Constraint 2: Success and Failure attributes
% If object of interest (j=2) and has effect (v=1),
% then success (w=1).
if (j == 2 && v == 1)
    if (w == 1)
        if (Pn(trace,13)~=0) Pn(trace,13) = 1; end
    else
        Pn(trace,13) = 0;
    end
end
% If object of interest (j=2) and not have effect (v=2),
% then fail (w=2).
if (j == 2 && v == 2)
    if (w == 2)
        if (Pn(trace,13)~=0) Pn(trace,13) = 1; end
    else
        Pn(trace,13) = 0;
    end
end

% If object not of interest (j=1) and no action (u=2),
% then success (w=1).
if (j == 1 && u == 2)
    if (w == 1)
        if (Pn(trace,13)~=0) Pn(trace,13) = 1; end
    else
        Pn(trace,13) = 0;
    end
end
% If object not of interest (j=1) and acted upon (u=1),
% then fail (w=2).
if (j == 1 && u == 1)
    if (w == 2)
        if (Pn(trace,13)~=0) Pn(trace,13) = 1; end
    else
        Pn(trace,13) = 0;
    end
end

% Constraint 3: Observability
% If object not presenting observation (k=2),
% then sensor does not detect (m=2).
if (k == 2)
    if (m == 2)
        if (Pn(trace,4)~=0) Pn(trace,4) = 1; end
    else
        Pn(trace,4) = 0;
    end
end

% Constraint 4: Sending data depends on detection
% If not detected (m=2),
% then data not sent (n=2).
if (m == 2)
    if (n == 2)
        if (Pn(trace,5)~=0) Pn(trace,5) = 1; end
    else
        Pn(trace,5) = 0;
    end
end

```

```

end
end

% Constraint 5: Receiving data depends on sending
% If not sent (p=2),
% then data not received (p=2).
if (n == 2)
    if (p == 2)
        if (Pn(trace,6)~=0) Pn(trace,6) = 1; end
    else
        Pn(trace,6) = 0;
    end
end

% Constraint 6: Merging data depends on receiving data
% If not received (p=2),
% then data not merged (q=2).
if (p == 2)
    if (q == 2)
        if (Pn(trace,7)~=0) Pn(trace,7) = 1; end
    else
        Pn(trace,7) = 0;
    end
end

% Constraint 7: Interpretation depends on merging data
% If data not merged (q=2),
% then interpret as unknown (r=3).
if (q == 2)
    if (r == 3)
        if (Pn(trace,8)~=0) Pn(trace,8) = 1; end
    else
        Pn(trace,8) = 0;
    end
end

% If data is merged (q=1),
% then do not interpret as unknown (q=3).
if (q == 1)
    if (r == 3)
        Pn(trace,8) = 0;
    end
end

% Constraint 8: Sending merged data depends on interpretation
% If not interpreted (q=2),
% then do not send merged data (s=2).
if (q == 2)
    if (s == 2)
        if (Pn(trace,9)~=0) Pn(trace,9) = 1; end
    else
        Pn(trace,9) = 0;
    end
end

% Constraint 9: Receiving merged depends on sending
% If not sent (s=2),
% then do not receive merged data (t=2).
if (s == 2)
    if (t == 2)
        if (Pn(trace,10)~=0) Pn(trace,10) = 1; end
    else
        Pn(trace,10) = 0;
    end
end

% Constraint 10: Getting hit depends on shooting
% If not shot (u=2),
% then not hit (v=2).
if (u == 2)
    if (v == 2)
        if (Pn(trace,12)~=0) Pn(trace,12) = 1; end
    else
        Pn(trace,12) = 0;
    end
end

% Constraint 11:
% If not receive merged data (t=2),
% then do not act (u=2).
if (t == 2)
    if (u == 2)
        if (Pn(trace,11)~=0) Pn(trace,11) = 1; end
    else
        Pn(trace,11) = 0;
    end
end

% Constraint 12:
% If interpreted as not an object of interest (r=1),
% then do not act (u=2).

```



```

traces=traces+1;
fprintf(' %3d; %5d; %8.5f; ',traces,trace,Pt(trace));
%
% fprintf(' P(i=%d)=%5.3f; ',i,Pn(trace,1));
% fprintf(' P(j=%d)=%5.3f; ',j,Pn(trace,2));
% fprintf(' P(k=%d)=%5.3f; ',k,Pn(trace,3));
% fprintf(' P(m=%d)=%5.3f; ',m,Pn(trace,4));
% fprintf(' P(n=%d)=%5.3f; ',n,Pn(trace,5));
% fprintf(' P(o=%d)=%5.3f; ',o,Pn(trace,6));
% fprintf(' P(q=%d)=%5.3f; ',q,Pn(trace,7));
% fprintf(' P(r=%d)=%5.3f; ',r,Pn(trace,8));
% fprintf(' P(s=%d)=%5.3f; ',s,Pn(trace,9));
% fprintf(' P(t=%d)=%5.3f; ',t,Pn(trace,10));
% fprintf(' P(u=%d)=%5.3f; ',u,Pn(trace,11));
% fprintf(' P(v=%d)=%5.3f; ',v,Pn(trace,12));
% fprintf(' P(w=%d)=%5.3f; ',w,Pn(trace,13));

fprintf(' %d; %5.3f; ',i,Pn(trace,1));
fprintf(' %d; %5.3f; ',j,Pn(trace,2));
fprintf(' %d; %5.3f; ',k,Pn(trace,3));
fprintf(' %d; %5.3f; ',m,Pn(trace,4));
fprintf(' %d; %5.3f; ',n,Pn(trace,5));
fprintf(' %d; %5.3f; ',o,Pn(trace,6));
fprintf(' %d; %5.3f; ',q,Pn(trace,7));
fprintf(' %d; %5.3f; ',r,Pn(trace,8));
fprintf(' %d; %5.3f; ',s,Pn(trace,9));
fprintf(' %d; %5.3f; ',t,Pn(trace,10));
fprintf(' %d; %5.3f; ',u,Pn(trace,11));
fprintf(' %d; %5.3f; ',v,Pn(trace,12));
fprintf(' %d; %5.3f; ',w,Pn(trace,13));

end
% Template 1: Identified as not an object of interest
if(i==1 && j==1 && k==1 && m==1 && n==1 && p==1 && q==1 ...
    && r==1 && s==1 && t==1 && u==2 && v==2 && w==1 )
    fprintf(' T1: Identified as not an object of interest');
end

% Template 2: Non-object, merged data not received
if(i==1 && j==1 && k==1 && m==1 && n==1 && p==1 && q==1 ...
    && r==1 && s==1 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T2: Non-object, merged data not received');
end

% Template 3: Non-object, merged data not sent
if(i==1 && j==1 && k==1 && m==1 && n==1 && p==1 && q==1 ...
    && r==1 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T3: Non-object, merged data not sent');
end

% Template 4: Non-object, interpreted as unknown
if(i==1 && j==1 && k==1 && m==1 && n==1 && p==1 && q==2 ...
    && r==3 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T4: Non-object, interpreted as unknown');
end

% Template 5: Non-object, data not received
if(i==1 && j==1 && k==1 && m==1 && n==1 && p==2 && q==2 ...
    && r==3 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T5: Non-object, data not received');
end

% Template 6: Non-object, data not sent
if(i==1 && j==1 && k==1 && m==1 && n==2 && p==2 && q==2 ...
    && r==3 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T6: Non-object, data not sent');
end

% Template 7: Non-object, not detected
if(i==1 && j==1 && k==1 && m==2 && n==2 && p==2 && q==2 ...
    && r==3 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T7: Non-object, not detected');
end

% Template 8: Non-object, not observable
if(i==1 && j==1 && k==2 && m==2 && n==2 && p==2 && q==2 ...
    && r==3 && s==2 && t==2 && u==2 && v==2 && w==1 )
    fprintf(' T8: Non-object, not observable');
end

% Template 9: Object receives effect
if(i==2 && j==2 && k==1 && m==1 && n==1 && p==1 && q==1 ...
    && r==2 && s==1 && t==1 && u==1 && v==1 && w==1 )
    fprintf(' T9: Object receives effect');
end

% Template 10: Object acted upon without effect
if(i==2 && j==2 && k==1 && m==1 && n==1 && p==1 && q==1 ...
    && r==2 && s==1 && t==1 && u==1 && v==2 && w==2 )
    fprintf(' T10: Object acted upon without effect');
end

```



```

for r = 1:3 % e8: interpretation :      column 8
    for s = 1:2 % e9: send merged data :      column 9
        for t = 1:2 % e10: receive merged data: column 10
            for u = 1:2 % e11: action :      column 11
                for v = 1:2 % e12: effect :      column 12
                    for w = 1:2 % e13: outcome :      column 13
                        ii = ii + 1;

% Present an observation (k=1) 90% of the time, column 3.
if (k == 1)
    Pn(ii,3) = .9;
else
    Pn(ii,3) = .1;
end

% If observation is presented (k=1) then detect (m=1) 99.9%,
% of the time, column 4.
if (k == 1)
    if (m==1)
        Pn(ii,4) = .999;
    else
        Pn(ii,4) = .001;
    end
end

% If detected (m=1) then send data (n=1) 99.9% of the time,
% column 5.
if (m == 1)
    if (n==1)
        Pn(ii,5) = .999;
    else
        Pn(ii,5) = .001;
    end
end

% If the data is sent (n=1), then
% receive the data (p=1) 99.9% of the time, column 6.
if (n == 1)
    if (p==1)
        Pn(ii,6) = .999;
    else
        Pn(ii,6) = .001;
    end
end

% If the data is received (p=1), then
% fuse the data (q=1) 99.9% of the time, column 7.
if (p == 1)
    if (q==1)
        Pn(ii,7) = .999;
    else
        Pn(ii,7) = .001;
    end
end

% If the data is merged (q=1) and is not of interest (j=1),
% then interpret as not of interest (r=1) 99.9% of the time,
% column 8.
if (q == 1 && j == 1)
    if (r==1)
        Pn(ii,8) = .999;
    else
        Pn(ii,8) = .001;
    end
end

% If the data is merged (q=1) and is of interest (j=2), then
% interpret as of interest (r=2) 99.9% of the time, column 8.
if (q == 1 && j == 2)
    if (r==2)
        Pn(ii,8) = .999;
    else
        Pn(ii,8) = .001;
    end
end

% If interpreted as of interest or not of interest (r = 1,2),
% then send the merged data (s=1) 99.9% of the time, column 9
if (r == 1 || r == 2)
    if (s==1)
        Pn(ii,9) = .999;
    else
        Pn(ii,9) = .001;
    end
end

% If the merged data is sent (s=1), then
% receive the data (t=1) 99.9% of the time, column 10.
if (s == 1)
    if (t == 1)

```

```
d
end
end
end
end
end
end
```

Calculate additional properties of the Object-to-Effect Model

- Script: `constraintPropertiesAction`

```
ii = 0; jj = 0; Ps = 0; Po = 0; Pu = 0; Ph = 0; Pf = 0;
for i = 1:2 % e1: problem state :      column 1
    for j = 1:2 % e2: object of interest : column 2
```

```

for k = 1:2 % e3: present observation : column 3
    for n = 1:2 % e4: detected : column 4
        for m = 1:2 % e5: send data : column 5
            for p = 1:2 % e6: receive data : column 6
                for q = 1:2 % e7: merge data : column 7
                    for r = 1:3 % e8: interpretation : column 8
                        for s = 1:2 % e9: send merged data : column 9
                            for t = 1:2 % e10: receive merged data: column 10
                                for u = 1:2 % e11: action : column 11
                                    for v = 1:2 % e12: effect : column 12
                                        for w = 1:2 % e13: outcome : column 13

                                            ii = ii + 1;
                                            if( Pt(ii) > 0 )
                                                jj=jj+1;
%         determine the probability of success;
                                                if ( w == 1)
                                                    Ps = Ps + Pt(ii);
                                                end
%         determine the probability of failure;
                                                if ( w == 2)
                                                    Po = Po + Pt(ii);
                                                end
%         determine the probability of interpret as object not of interest;
                                                if ( r == 1 )
                                                    Pf = Pf + Pt(ii);
                                                end
%         determine the probability of interpret as object of interest;
                                                if ( r == 2 )
                                                    Ph = Ph + Pt(ii);
                                                end
%         determine the probability of interpret as unknown;
                                                if ( r == 3)
                                                    Pu = Pu + Pt(ii);
                                                end

                                            end

                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end

fprintf('Probability of success: %6.4f\n',Ps);
fprintf('Probability of failure: %6.4f\n',Po);
fprintf('Probability of ID as not an object of interest: %6.4f\n',Pf);
fprintf('Probability of ID as object of interest : %6.4f\n',Ph);
fprintf('Probability of ID as unknown : %6.4f\n',Pu);

```

Published with MATLAB® R2014b

Contents

- Function [p] = trimBranchAction(P,eventnum)
- Determine the number of rows and columns based on the matrix, P
- Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.

```
function [p] = trimBranchAction(P,eventnum)
```

Function [p] = trimBranchAction(P,eventnum)

This function operates on a probability matrix, with each decision based on any number of alternatives defined by the vector eventnum. The algorithm eliminates probabilities that have been constrained from occurring (p=0), prorates the probability of the remaining branches by adjusting the remaining alternatives, and returns the resulting trimmed probability matrix.

```

% P is the probability matrix
% eventnum: matrix that identifies the number of alternatives of each event

% Written by John Quartuccio
% 21 Apr 2017
% 28 Jul 2019 updated for general numbers of alternatives per event
% 24 Aug 2019 finalized for publication

```

Determine the number of rows and columns based on the matrix, P

```
numrows = length(P); % the number of rows, each representing a trace
numcols = size(P,2); % the number of columns, each representing an event

%% Zero out probabilities for the entire row (trace) if any zero is found
%% in any column (event) of the trace
% for m = 1:numrows % m is index for each row (trace)
%   for n = 1:numcols % n is index for each column (event)
%       if (P(m,n) == 0)
%           P(m,:) = 0;
%       %
%       end % end of if
%   end % end of column index, n (event)
% end % end of row index, m (trace)
```

Step through the probability matrix from right to left to determine constrained traces and prorating adjustment.

Initialize values for the step size and number of events per step. The step size indicates the number of alternatives associated with the event

```
step = numrows; % reset the step index
neps = 1; % initialize the number of events per step
%
for n = numcols :-1: 1 % index, n is the column number
    step = step/eventnum(n); % derive the number of steps in the event column
    neps = neps*eventnum(n); % derive the number of events per step
    % fprintf('\n\nColumn %d:\n',n) % debug
    % fprintf('    number of steps: %d\n',step) % debug
    % fprintf('    number of events per step: %d\n',neps) % debug
    %
    m = 0; % reset the row index, m
    for s = 1:step % index, s, for the step associated with each set of
        % events, eventnum(n)
        stepsum = 0; % reset the step sum of probabilities
        numnon0 = 0; % reset the number of events with non-zero probability
        num0 = 0; % reset the number of events with zero probability
        % fprintf('    step: %d\n',s) % debug
        for e = 1:neps
            m = m + 1;
            stepsum = stepsum + P(m,n);
            if (P(m,n)==0)
                num0 = num0 + 1;
                row0 = m;
            % fprintf('    zero row index: %d\n',row0) % debug
            end % end if
            if (P(m,n)~=0)
                numnon0 = numnon0 + 1;
            end % end if
        end % end of event block: e = 1:neps

        % Determine the portion of probability to adjust each nonzero value
        Padj = (1 - stepsum*eventnum(n)/neps) * neps / eventnum(n) / numnon0;
        % fprintf('    step sum of p: %5.3f\n',stepsum) % debug
        % fprintf('    number non-zero: %d\n',numnon0) % debug
        % fprintf('    total p in step: %d\n',(1*neps/eventnum(n))) % debug
        % fprintf('    adjusted value: %5.3f\n', Padj) % debug

        % If sum of probabilities is zero, adjust the "upstream" set to zero
        if (stepsum == 0)
            for e = 1:neps
                P((m-e+1),(n-1)) = 0;
            end % end of event block
        end % end if (stepsum == 0)

        % Allocate residual probability to non-zero values
        if (int16(stepsum*1000) ~= int16(1*neps/eventnum(n)*1000) ...
            && stepsum ~= 0)
            for e = 1:neps
                if (P((m-e+1),n) ~= 0)
                    P(m-e+1,n) = P(m-e+1,n) + Padj;
                end % end if
            end % end of event block
        end % end if (stepsum ~= 1)

    end % end of step
end % end of pass

p=P;
```

```
end %end of function
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C:

Derivation of the Design Structure Matrix (DSM)

The following sections provide the derivation of the DSM, comparison analysis, and cluster analysis for the Decision Model and the Object-to-Effect Model.

C.1 Decision Model

C.1.1 DSM Derived from the Decision Model

This script builds an N-squared diagram, based on the directed graph of Monterey Phoenix Output

Contents

- Script: MPbuildN2.m
- Clear all variables and screens
- Add the script location to the path
- Enter the filename and open the output files
- Open the output files
- Indicate whether to include "Say" events in the N-squared diagram.
- Read the .json file
- Determine the number of traces, events, and links
- Derive the event labels as a structured variable
- Find the list of unique event names,
- Build the event edges as a structured variable
- Derive the cumulative adjacency matrices
- Print the N-squared diagram that consists of the event list and cM
- Build the DSM.
- Execute DSM Clustering script

Called Functions

- MPdsm
- MPfindEdges
- MPfindEventList
- MPfindEvents
- MPmatrix
- MPOpenOutput
- MPprintN2
- MPreadWNGv3

Script: MPbuildN2.m

The script is compatible with MP-Firebird or MP-Gryphon, versions 3.5 and 4. The script operates on MP .wng output of code and graphs.

```
% Written by
% John Quartuccio

% 30 SEP 2016 executable script readWNGv3

% 12 JAN 2018 break-out of sub-functions

% 29 Jul 2018 Updated to v3 of MP

% 08 AUG 2018 Updated to include input and output

% 22 AUG 2018 Updated comments

% 10 NOV 2019 Updated for publication
%
% The script reads the .json file and derives a single list of unique
% events. Then tabulates all event interactions, with event rows as the
% source (from) and event columns as the target (to) of the interactions.
%
% Note: the first few characters may contain an encoding artifact: "i»¿"
% These characters need to be deleted in order to read by Matlab.
%
% The convention used has been to open the <filename>.wng file in Matlab,
% and then delete the characters prior to "{"entities":..." and save as
% <filename>.json
%
% The following functions need to exist on the Matlab path, listed here in
% order of occurrence:

%     MPOpenOutput.m      Opens the output files, based on <filename>

%     MPreadWNGv3.m       Reads the .wng file in MP version 3 format

%     MPfindEvents.m      Finds the events

%     MPfindEventList.m   Builds the unique list of events
```

```
% MPfindEdges.m      Finds the edges
% MPmatrix.m         Derives the cumulative matrix of all edges
% MPprintN2.m        Prints the N-squared diagram to .csv
% MPdsm.m            Builds the Design Structure Matrix based on the
%                    N-squared diagram for further cluster analysis
% MPanalyzeDSM.m     Executes the cluster analysis
```

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;
path(oldpath,'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename and open the output files

```
filename = 'decisionModelScope1';
```

Open the output files

```
MPOpenOutput;
```

Indicate whether to include "Say" events in the N-squared diagram.

```
doSay = false;% excludes "Say" events
```

Read the .json file

The parser returns a string variable comprised of the entire .json file. This function populates a structured variable, MP, as follows:

```
% MP.entities.traces.<guid of trace>.guid    <guid of trace>
%                                           .layout    <SEQUENCE>
%                                           <FORCE>
%                                           <SWIMLANE>
%                                           .marked    <logical>
%                                           .nodes     <guid of nodes>
%                                           .links     <guid of links>

% MP.entities.nodes.<guid of node> .guid    <guid of node>
%                                           .name      <user-defined event name>
%                                           .type      <ROOT>
%                                           <COMPOSITE>
%                                           <ATOM>
%                                           <SAY>
%                                           .x         <location>
%                                           .y         <location>
%                                           .collapsed <logical>
%                                           .hidden    <logical>
% MP.entities.links.<guid of link> .guid    <guid of link>
```

```
%
%           .type      <IN>
%
%           <FOLLOWS>
%
%           <NAMED>
%
%           .text      <user-defined text>
%
%           .source     <guid of source event>
%
%           .target     <guid of target event>
%
%           .points     <x,y location>
%
%           .curviness  <value>

MP = MPreadWNgv3(filename);
```

Determine the number of traces, events, and links

```
numTraces = length(struct2cell(MP.entities.traces));
numEvents = length(struct2cell(MP.entities.nodes));
numEdges  = length(struct2cell(MP.entities.links));
```

Derive the event labels as a structured variable

For each trace, n, and each event, i, as follows:

```
%      Events(n).name{i} : the user-defined name of each event
%
%      Events(n).type{i} : the type follows the following description:
%      <ROOT>, <ATOM>, <COMPOSITE>, <SAY>
%
%      Events(n).guid(i) : the graph ID for each event in each trace
%
%      Events(n).ID(i)   : the ID is the reference id for each trace
%
%      Events(n).mapID(i) : empty field that used to map each event
%                        label to a list of unique names
Events = MPfindEvents(MP);
```

Find the list of unique event names,

Identify the labels and map the index of the list to the structured array: Events.mapID

```
[Events, eventList] = MPfindEventList(Events, doSay);
```

Build the event edges as a structured variable

for each trace, n, and each event, i, with both inclusion and precedence relationships, indicating the "from" and "to" directions as follows:

```
%      Edges(n).inclusion(i).to
%      Edges(n).inclusion(i).from
%
%      Edges(n).precedence(i).to
%      Edges(n).precedence(i).from
%
%      Edges(n).named(i).to
%      Edges(n).named(i).from
Edges = MPfindEdges(MP, Events);
```

Derive the cumulative adjacency matrices

```
%      cM.all       : sum of all edges
%
%      cM.inclusion  : sum of inclusion edges
%
%      cM.precedence : sum of precedence edges
%
%      cM.named     : sum of named edges
```

```
cM = MPmatrix(Edges,eventList);
```

Print the N-squared diagram that consists of the event list and cM

```
MPprintN2(eventList, cM, fout, fcsv);
```

N-squared diagram:

		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
ROOT	Leader(1)	0	0	0	4	8	8	8	0	0	0	0	0	8	4	4	4	0	0
ROOT	Subordinates(2)	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0
ROOT	Environment(3)	0	0	0	0	0	0	0	0	0	4	8	8	0	0	0	0	4	0
COMPOSITE	recognizeenvironment(4)	0	0	0	0	4	0	0	0	5	0	0	0	0	0	0	0	0	1
COMPOSITE	receiveinput(5)	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	decidecorrectly(6)	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	actcorrectly(7)	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0
COMPOSITE	Subordinate(8)	0	0	0	6	0	0	0	0	10	0	0	0	6	0	0	0	0	2
COMPOSITE	communicateobservation(9)	0	0	0	0	8	0	0	0	0	0	0	0	0	2	0	0	0	0
COMPOSITE	notcontainproblem(10)	0	0	0	4	0	0	0	0	0	0	4	0	4	0	0	0	0	0
COMPOSITE	desiredresult(11)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	containproblem(12)	0	0	0	6	0	0	0	0	0	0	4	0	10	0	0	0	4	0
COMPOSITE	notrecognizeenvironment(13)	0	0	0	0	4	0	0	0	5	0	0	0	0	4	0	0	0	1
COMPOSITE	notreceiveinput(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0
COMPOSITE	notdecidecorrectly(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0
COMPOSITE	notactcorrectly(16)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
COMPOSITE	notdesiredresult(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	notcommunicateobservation(18)	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0

Build the DSM.

The adjacency matrix, cM, aligns interactions FROM the row as the "source" and TO the column as the "target." This is known as the input-row (IR) convention. Conversely, the design structure matrix (DSM) uses the input column (IC) convention and so the DSM is a transpose of cM. Also, the DSM does not calculate the accumulated number of interactions, it just indicates that a connection exists, as a one (1) and that if no interface exists, indicated as a "0".

```
DSM = MPdsm(cM.all);
```

Execute DSM Clustering script

```
MPanalyzeDSM;
```

Published with MATLAB® R2014b

```
function M = MPdsm(A)
M = A.';
for i = 1 : length(A)
    for j = 1 : length(A)
        if (M(i,j) > 0) M(i,j) = 1; end
        if (i == j) M(i,j) = 1; end
    end
end
```

Published with MATLAB® R2014b

```
function Edges = MPfindEdges(MP, Events)
% define a structured variable for each trace, n, for the title, type,
% and refID of each event, i, in the trace
%
numTraces = length(struct2cell(MP.entities.traces));
numEdges = length(struct2cell(MP.entities.links));
edgeGuid = char(fieldnames(MP.entities.links));

Edges = struct('inclusion',[],'precedence',[],'named',[]);
Edges.inclusion = struct('to',[],'from',[]);
Edges.precedence = struct('to',[],'from',[]);
Edges.named = struct('to',[],'from',[]);

%
```

```

%
% find the guid for each edge, n, and then relate this to the Edges
% structured matrix
%   currentEdge.guid <guid>
%   .type "IN", "FOLLOWS", or "NAMED"
%   .text <user-defined text>
%   .source <guid of source>
%   .target <guid of target>
%   Events .guid
%
iSourceIn = 0; iTargetIn = 0;
iSourcePrec = 0; iTargetPrec = 0;
iSourceNamed = 0; iTargetNamed = 0;
for n = 1 : numEdges
    currentEdge = eval(strcat('MP.entities.links.',edgeGuid(n,:)));
    currentSource = valid_field(currentEdge.source);
    currentTarget = valid_field(currentEdge.target);
%
%
% find the name for the target and source, based on the guid,
%
for i = 1 : numTraces
    for j = 1 : length(Events(i).guid)
%
        if (strcmp(currentSource, Events(i).guid{j}))
            if (strcmp('IN',currentEdge.type))
                iSourceIn = iSourceIn + 1;
                Edges.inclusion.from{iSourceIn} = Events(i).mapID{j};
            end % (if IN)

            if (strcmp('FOLLOWS',currentEdge.type))
                iSourcePrec = iSourcePrec + 1;
                Edges.precedence.from{iSourcePrec} = Events(i).mapID{j};
            end % (if FOLLOWS)

            if (strcmp('NAMED',currentEdge.type))
                iSourceNamed = iSourceNamed + 1;
                Edges.named.from{iSourceNamed} = Events(i).mapID{j};
            end % (if IN)
        end % (if current source equal to guid)
%
        if (strcmp(currentTarget, Events(i).guid{j}))
            if (strcmp('IN',currentEdge.type))
                iTargetIn = iTargetIn + 1;
                Edges.inclusion.to{iTargetIn} = Events(i).mapID{j};
            end % (if IN)

            if (strcmp('FOLLOWS',currentEdge.type))
                iTargetPrec = iTargetPrec + 1;
                Edges.precedence.to{iTargetPrec} = Events(i).mapID{j};
            end % (if FOLLOWS)

            if (strcmp('NAMED',currentEdge.type))
                iTargetNamed = iTargetNamed + 1;
                Edges.named.to{iTargetNamed} = Events(i).mapID{j};
            end % (if NAMED)
        end % (if current source equal to guid)

    end % (Events within the trace)
end % (number of traces)

end % end of for loop (numEdges)

end % end of function (MPFindEdges)

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
    if ~isletter(str(1))
        str = ['alpha_' str];
    end
    str(~isletter(str) & ~('0' <= str & str <= '9')) = '_';
end

```



```

function [Events, eventList] = MPfindEventList(Events, doSay)
% MPbuildUnique finds the unique list of event names and provide a
% reference to the index of the unique list.
%
%
% find the number of traces, based on the Events structured array
numTraces = length(Events);

%
% for each type of event, (root, composite, atom, or say), find the unique
% names used
uniqueEvents = struct('root',[], 'composite',[], 'atom',[], 'say',[]);
ir = 0; ic = 0; ia = 0; isay = 0;
for n = 1:numTraces
    for i = 1:length(Events(n).name)
        if (strcmp(Events(n).type(i), 'ROOT'))
            if(ir == 1)
                ir = ir + 1;
                uniqueEvents.root{ir} = Events(n).name{i};
            else
                flag = 0;
                for j = 1:ir
                    if (strcmp(uniqueEvents.root{j}, Events(n).name{i}))
                        flag = 0;
                    else
                        flag = flag + 1;
                    end
                end
                if (flag == ir)
                    ir = ir + 1;
                    uniqueEvents.root{ir} = Events(n).name{i};
                end
            end
        elseif(strcmp(Events(n).type(i), 'COMPOSITE'))
            if(ir == 1)
                ic = ic + 1;
                uniqueEvents.composite{ic} = Events(n).name{i};
            else
                flag = 0;
                for j = 1:ic
                    if (strcmp(uniqueEvents.composite{j}, Events(n).name{i}))
                        flag = 0;
                    else
                        flag = flag + 1;
                    end
                end
                if (flag == ic)
                    ic = ic + 1;
                    uniqueEvents.composite{ic} = Events(n).name{i};
                end
            end
        elseif(strcmp(Events(n).type(i), 'ATOM'))
            if(ia == 1)
                ia = ia + 1;
                uniqueEvents.atom{ia} = Events(n).name{i};
            else
                flag = 0;
                for j = 1:ia
                    if (strcmp(uniqueEvents.atom{j}, Events(n).name{i}))
                        flag = 0;
                    else
                        flag = flag + 1;
                    end
                end
                if (flag == ia)
                    ia = ia + 1;
                    uniqueEvents.atom{ia} = Events(n).name{i};
                end
            end
        elseif(strcmp(Events(n).type(i), 'SAY'))
            if(isay == 1)
                isay = isay + 1;
                uniqueEvents.say{ir} = Events(n).name{i};
            end
        end
    end
end

```

```

        else
            flag = 0;
            for j = 1:isay
                if (strcmp(uniqueEvents.say(j),Events(n).name(i)))
                    flag = 0;
                else
                    flag = flag + 1;
                end
            end
            if (flag == isay)
                isay = isay + 1;
                uniqueEvents.say{isay} = Events(n).name(i);
            end
        end
    end
end
end
itotal = ir + ic + ia + isay;

eventList = struct('name',[],'type',[]);
i = 0;
for j = 1:length(uniqueEvents.root)
    i = i + 1;
    eventList.name{i} = uniqueEvents.root{j};
    eventList.type{i} = 'ROOT';
end

for j = 1:length(uniqueEvents.composite)
    i = i + 1;
    eventList.name{i} = uniqueEvents.composite{j};
    eventList.type{i} = 'COMPOSITE';
end

for j = 1:length(uniqueEvents.atom)
    i = i + 1;
    eventList.name{i} = uniqueEvents.atom{j};
    eventList.type{i} = 'ATOM';
end

if (doSay)
    for j = 1:length(uniqueEvents.say)
        i = i + 1;
        eventList.name{i} = uniqueEvents.say{j};
        eventList.type{i} = 'SAY';
    end
end

for n = 1:numTraces
    for i = 1: length(Events(n).name)
        for j = 1:length(eventList.name)
            if strcmp(Events(n).name{i}, eventList.name{j})
                Events(n).mapID{i} = j;
            end
        end
    end
end
end
end
end

```

Published with MATLAB® R2014b

```

function Events = MPfindEvents(MP)
% define a structured variable for the events in each trace, n, including
% the guid, name, type, and index of each event, i, in the trace
%
% determine the number of traces
numTraces = length(struct2cell(MP.entities.traces));
traceGuid = char(fieldnames(MP.entities.traces));

%
% define the Labels variable
%
Events = struct('name',[],'type',[],'guid',[],'ID',[],'mapID',[]);

%

```

```

%
% find the guid for each node, i, in each trace, n and then relate this to
% the Events structured matrix
%
for n = 1 : numTraces
    currentTrace = eval(strcat('MP.entities.traces.',traceGuid(n,:)));
    numEventsInTrace = length(currentTrace.nodes);
    for i = 1 : numEventsInTrace
        c = valid_field(currentTrace.nodes{i});
        Events(n).name{i} = eval(strcat('MP.entities.nodes.',c,'.name'));
        Events(n).type{i} = eval(strcat('MP.entities.nodes.',c,'.type'));
        Events(n).guid{i} = c;
        Events(n).ID{i} = i;
    end % end of for loop (numEventsInTrace)
end % end of for loop (numTraces)

end % end of function (MPfindEvents)

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
if ~isletter(str(1))
    str = ['alpha_' str];
end
str(~isletter(str) & ~{'0' <= str & str <= '9'}) = '_';
end

```

Published with MATLAB® R2014b

```

function cM = MPmatrix(Edges,eventList)
% John Quartuccio
% 4 Jan 2018
% revised 7 Aug 2018
% Function MPmatrix(Edges,eventList) returns the cumulative adjacency
% matrix for inclusion, precedence, and named edges.
%
% Edges.<type>.<from/to> captures the reference ID consistent with the
% order of the index of the eventList
%
% Initialize the cumulative adjacency matrices
cM.all = zeros(length(eventList.name),length(eventList.name));
cM.inclusion = cM.all; cM.precedence = cM.all; cM.named = cM.all;

for i = 1 : length(Edges.inclusion.from)
    cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) = ...
        cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) + 1;
end % (end of for loop of inclusion edges)

for i = 1 : length(Edges.precedence.from)
    cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) = ...
        cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) + 1;
end % (end of for loop of precedence edges)

for i = 1 : length(Edges.named.from)
    cM.named(Edges.named.from{i},Edges.named.to{i}) = ...
        cM.named(Edges.named.from{i},Edges.named.to{i}) + 1;
end % (end of for loop of named edges)

cM.all = cM.inclusion + cM.precedence + cM.named;

end % (end of MPmatrix function)

```

Published with MATLAB® R2014b

```
%
% open the output files
fout = fopen(strcat(filename, '.txt'), 'w+');
fcsv = fopen(strcat(filename, '.csv'), 'w+');
ftex = fopen(strcat(filename, '.tex'), 'w+');
```

Published with MATLAB® R2014b

This function prints the N2 diagram to the screen and output files

```
function MPprintN2(eventList, cM, fout, fcsv)
%
% Print the cumulative matrix, cMa
%
% for the screen output and the text files, determine the width of the
% column on the basis of the length of event names.
len = longestName(eventList.name);
% if (len>16), len = 16; end;
sep = 6;
colstringformat = strcat('%', num2str(len), 's');
colnumformat = strcat('%', num2str(sep), 'd');
colindexformat = strcat('%', num2str(sep-4), 's(%2s)');
rowstringformat = strcat('%', num2str(len), 's(%2s)');

skip = ' ';
%
fprintf( ' \n N-squared diagram:\n');
fprintf( fout, '\n N-squared diagram:\n');
%
% Print the column headers
% fprintf( ' %16s ', ' ');
% fprintf( colstringformat, ' ');
% fprintf(fout, ' %16s ', ' ');
% fprintf(fout, colstringformat, ' ');
% fprintf(fcsv, '%s, %s, %s, ', 'Event type', 'Event name', 'index');
%
for i = 1:length(eventList.name)
% fprintf( colstringformat, eventList.name{i});
% fprintf(fout, colstringformat, eventList.name{i});
% fprintf(fcsv, '%s, ', eventList.name{i});
end % ( end of print column headers)
%
% fprintf( ' \n');
% fprintf(fout, '\n');
% fprintf(fcsv, '\n');

fprintf( strcat('%14s ', num2str(len), 's '), skip, skip);
fprintf(fout, strcat('%16s ', num2str(len), 's '), skip, skip);
fprintf(fcsv, '\n , , ');
%
for i = 1:length(eventList.name)
% fprintf( colindexformat, skip, num2str(i));
% fprintf(fout, colindexformat, skip, num2str(i));
% fprintf(fcsv, ', (%d)', i);
end % ( end of print column index)
%
% fprintf( ' \n');
% fprintf(fout, '\n');
% fprintf(fcsv, '\n');

%
% Print out the type, name, and values of the N-squared diagram
% m: index for each row of the cumulative matrix
% n: index for the column of the cumulative matrix
for m = 1 : length(eventList.name);
% fprintf( ' %10s ', eventList.type{m});
% fprintf( rowstringformat, eventList.name{m}, num2str(m));
% fprintf(fout, ' %10s ', eventList.type{m});
% fprintf(fout, rowstringformat, eventList.name{m}, num2str(m));
% fprintf(fcsv, '%s, %s, (%s)', ...
eventList.type{m}, eventList.name{m}, num2str(m));
%
end
```

```

    for n = 1 : length(eventlist.name);
        fprintf(    colnumformat, cM.all(m,n));
        fprintf(fout, colnumformat, cM.all(m,n));
        if (cM.all(m,n) == 0)
            fprintf(fcsv, '%s', skip);
        else
            fprintf(fcsv, '%d', cM.all(m,n));
        end % (end of if)
        end % (end of print each column of current row)
        fprintf('\n'); fprintf(fout, '\n'); fprintf(fcsv, '\n');
    end % (end of print rows)
end % (end of function)

function len = longestName(name)
% returns the longest name of the structured matrix, name.
    len = 4;
    for i = 1 : length(name)
        x = length(name{i});
        if (x > len)
            len = x;
        end % (end of if)
    end % (end of for)
end % (end of function)

```

Published with MATLAB® R2014b

```

function MP = MPreadWNGv3(filename)
% John Quartuccio
% 30 Sep 2016
% updated 24 Apr 2017
% updated 04 Jan 2018
% updated 28 Jul 2018 to MP version 3, output by Gryphon and Firebird
%
% This script reads the .wng file in .json format as produced as file by
% executing Monterey Phoenix through MP-Firebird Analyzer at
% nps.firebird.edu
%
% The routine employs the parse_json.m function,
% downloaded 30 Sep 2016 from
% https://www.mathworks.com/matlabcentral/fileexchange/
% 23393--another--json-parser
% Other parsers available at https://json.org/
%
%
%
fjson = fopen(strcat(filename, '.json'));
MP = parse_json(fscanf(fjson, '%s'));
fclose(fjson);

```

Published with MATLAB® R2014b

C.1.2 Comparison of Decision Model Results at Scope 1 to 2

This script compares DSM from two sets of MP output files

Contents

- Script: RunCompareDSM
- Clear all variables and screens
- Add the script location to the path
- Enter the filename of the first file
- Build the N2 Diagram for the first file
- Enter the filename of the second file
- Build the N2 Diagram for the second file
- Compare the DSMs

Script: RunCompareDSM

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;  
path(oldpath, 'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename of the first file

```
filename = 'decisionModelScope1';  
filenameA = filename;
```

Build the N2 Diagram for the first file

```
MPbuildN2;  
DSMa = MPdsm(cM.all);
```

N-squared diagram:

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
ROOT	Leader(1)	0	0	0	4	8	8	8	0	0	0	0	8	4	4	4	0	0
ROOT	Subordinates(2)	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0
ROOT	Environment(3)	0	0	0	0	0	0	0	0	4	8	8	0	0	0	0	4	0
COMPOSITE	recognizeenvironment(4)	0	0	0	0	4	0	0	0	5	0	0	0	0	0	0	0	1
COMPOSITE	receiveinput(5)	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	decidecorrectly(6)	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0
COMPOSITE	actcorrectly(7)	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0
COMPOSITE	Subordinate(8)	0	0	0	6	0	0	0	0	10	0	0	0	6	0	0	0	2
COMPOSITE	communicateobservation(9)	0	0	0	0	8	0	0	0	0	0	0	0	2	0	0	0	0
COMPOSITE	notcontainproblem(10)	0	0	0	4	0	0	0	0	0	4	0	4	0	0	0	0	0
COMPOSITE	desiredresult(11)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	containproblem(12)	0	0	0	6	0	0	0	0	0	4	0	10	0	0	0	4	0
COMPOSITE	notrecognizeenvironment(13)	0	0	0	0	4	0	0	0	5	0	0	0	4	0	0	0	1
COMPOSITE	notreceiveinput(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0
COMPOSITE	notdecidecorrectly(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0
COMPOSITE	notactcorrectly(16)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0
COMPOSITE	notdesiredresult(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	notcommunicateobservation(18)	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0

Enter the filename of the second file

```
filename = 'decisionModelScope2';  
filenameB = filename;
```

Build the N2 Diagram for the second file

read the file;

```
MPOpenOutput;  
MP = MPreadWGV3(filename);  
% Determine the number of traces, events, and links  
numTraces = length(struct2cell(MP.entities.traces));  
numEvents = length(struct2cell(MP.entities.nodes));
```

```

numEdges = length(struct2cell(MP.entities.links));
% Find the events
Events = MPfindEvents(MP);
% Map events to the event list
for n = 1:numTraces
    for i = 1:length(Events(n).name)
        for j = 1:length(eventList.name)
            if strcmp(Events(n).name{i}, eventList.name{j})
                Events(n).mapID{i} = j;
            end
        end
    end
end
% Build the event edges as a structured variable
Edges = MPfindEdges(MP, Events);
% Derive the cumulative adjacency matrices
cM = MPmatrix(Edges,eventList);
% Print the N-squared diagram that consists of the event list and cM
MPprintN2(eventList, cM, fout, fcsv);
% Find the DSM for the second set
DSMb = MPdsm(cM.all);

```

N-squared diagram:

		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
ROOT	Leader(1)	0	0	0	24	48	48	48	0	0	0	0	0	40	16	16	16	0	0
ROOT	Subordinates(2)	0	0	0	0	0	0	128	0	0	0	0	0	0	0	0	0	0	0
ROOT	Environment(3)	0	0	0	0	0	0	0	0	24	48	40	0	0	0	0	0	16	0
COMPOSITE	recognizeenvironment(4)	0	0	0	24	0	0	0	40	0	0	0	0	0	0	0	0	0	24
COMPOSITE	receiveinput(5)	0	0	0	0	48	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	decidecorrectly(6)	0	0	0	0	0	48	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	actcorrectly(7)	0	0	0	0	0	0	0	0	0	48	0	0	0	0	0	0	0	0
COMPOSITE	Subordinate(8)	0	0	0	64	0	0	0	80	0	0	0	64	0	0	0	0	0	48
COMPOSITE	communicateobservation(9)	0	0	0	64	0	0	0	0	0	0	0	0	28	0	0	0	0	0
COMPOSITE	notcontainproblem(10)	0	0	0	36	0	0	0	0	0	24	0	36	0	0	0	0	0	0
COMPOSITE	desiredresult(11)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	containproblem(12)	0	0	0	52	0	0	0	0	0	24	0	68	0	0	0	16	0	0
COMPOSITE	notrecognizeenvironment(13)	0	0	0	24	0	0	0	40	0	0	0	0	16	0	0	0	24	0
COMPOSITE	notreceiveinput(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	16	0	0	0	0
COMPOSITE	notdecidecorrectly(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	0	0	0
COMPOSITE	notactcorrectly(16)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	0	0
COMPOSITE	notdesiredresult(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	notcommunicateobservation(18)	0	0	0	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0

Compare the DSMs

```

flag = 0;
Diff = DSMb - DSMa;
fprintf('\nSum of DSM edges from %s is %d\n', filenameA, sum(sum(DSMa)));
fprintf('Sum of DSM edges from %s is %d\n', filenameB, sum(sum(DSMb)));

for i = 1 : length(Diff)
    for j = 1 : length(Diff)
        if (Diff(i,j)~= 0)
            flag = 1;
            fprintf('i = %d, j = %d:', i,j);
            if (Diff(i,j) > 0)
                fprintf(' %s has relationship from %s to %s not shared by %s\n',...
                    filenameB, eventList.name{j}, eventList.name{i}, filenameA);
            end
            if (Diff(i,j) < 0)
                fprintf(' %s has relationship from %s to %s not shared by %s\n',...
                    filenameA, eventList.name{j}, eventList.name{i}, filenameB);
            end
        end
    end
end

if (flag == 0)
    fprintf('All relationships between these models are consistent.\n\n');
end

```

Sum of DSM edges from decisionModelScope1 is 58
Sum of DSM edges from decisionModelScope2 is 58
All relationships between these models are consistent.

C.1.3 Cluster Analysis of the Decision Model

This script builds an N-squared diagram, based on the directed graph of Monterey Phoenix Output

Contents

- [Script: MPbuildN2.m](#)
- [Clear all variables and screens](#)
- [Add the script location to the path](#)
- [Enter the filename and open the output files](#)
- [Open the output files](#)
- [Indicate whether to include "Say" events in the N-squared diagram.](#)
- [Read the .json file](#)
- [Determine the number of traces, events, and links](#)
- [Derive the event labels as a structured variable](#)
- [Find the list of unique event names,](#)
- [Build the event edges as a structured variable](#)
- [Derive the cumulative adjacency matrices](#)
- [Print the N-squared diagram that consists of the event list and cM](#)
- [Build the DSM.](#)
- [Execute DSM Clustering script](#)

Script: MPbuildN2.m

The script is compatible with MP-Firebird or MP-Gryphon, versions 3.5 and 4. The script operates on MP .wng output of code and graphs.

```
% Written by
% John Quantuccio

% 30 SEP 2016 executable script readWNGv3

% 12 JAN 2018 break-out of sub-functions

% 29 Jul 2018 Updated to v3 of MP

% 08 AUG 2018 Updated to include input and output

% 22 AUG 2018 Updated comments

% 10 NOV 2019 Updated for publication
%
% The script reads the .json file and derives a single list of unique
% events. Then tabulates all event interactions, with event rows as the
% source (from) and event columns as the target (to) of the interactions.
%
% Note: the first few characters may contain an encoding artifact: "ï»¿"
% These characters need to be deleted in order to read by Matlab.
%
% The convention used has been to open the <filename>.wng file in Matlab,
% and then delete the characters prior to "{"entities":..." and save as
% <filename>.json
%
% The following functions need to exist on the Matlab path, listed here in
% order of occurrence:

%   MPOpenOutput.m      Opens the output files, based on <filename>
%
%   MPreadWNGv3.m       Reads the .wng file in MP version 3 format
%
%   MPfindEvents.m       Finds the events
%
%   MPfindEventList.m    Builds the unique list of events
%
%   MPfindEdges.m        Finds the edges
%
%   MPmatrix.m           Derives the cumulative matrix of all edges
%
%   MPprintN2.m          Prints the N-squared diagram to .csv
%
%   MPdsm.m              Builds the Design Structure Matrix based on the
%                        N-squared diagram for further cluster analysis
%
%   MPanalyzeDSM.m       Executes the cluster analysis
```

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;
path(oldpath, 'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename and open the output files

```
filename = 'decisionModelScope1';
```

Open the output files

```
MPOpenOutput;
```

Indicate whether to include "Say" events in the N-squared diagram.

```
doSay = true; % includes "Say" events
```

```
doSay = false;% excludes "Say" events
```

Read the .json file

The parser returns a string variable comprised of the entire .json file. This function populates a structured variable, MP, as follows:

```
% MP.entities.traces.<guid of trace>.guid    <guid of trace>
%                                           .layout    <SEQUENCE>
%                                           <FORCE>
%                                           <SWIMLANE>
%                                           .marked    <logical>
%                                           .nodes     <guid of nodes>
%                                           .links     <guid of links>

% MP.entities.nodes.<guid of node> .guid    <guid of node>
%                                           .name      <user-defined event name>
%                                           .type      <ROOT>
%                                           <COMPOSITE>
%                                           <ATOM>
%                                           <SAY>
%                                           .x         <location>
%                                           .y         <location>
%                                           .collapsed <logical>
%                                           .hidden    <logical>
% MP.entities.links.<guid of link> .guid    <guid of link>
%                                           .type      <IN>
%                                           <FOLLOWS>
%                                           <NAMED>
%                                           .text      <user-defined text>
%                                           .source    <guid of source event>
%                                           .target    <guid of target event>
%                                           .points    <x,y location>
%                                           .curviness <value>

MP = MPreadWNGv3(filename);
```

Determine the number of traces, events, and links

```
numTraces = length(struct2cell(MP.entities.traces));
numEvents = length(struct2cell(MP.entities.nodes));
numEdges  = length(struct2cell(MP.entities.links));
```

Derive the event labels as a structured variable

For each trace, n, and each event, i, as follows:

```
%      Events(n).name{i} : the user-defined name of each event

%      Events(n).type{i} : the type follows the following description:
%      <ROOT>, <ATOM>, <COMPOSITE>, <SAY>

%      Events(n).guid(i) : the graph ID for each event in each trace

%      Events(n).ID(i)   : the ID is the reference id for each trace

%      Events(n).mapID(i) : empty field that used to map each event
%                        label to a list of unique names
Events = MPfindEvents(MP);
```

Find the list of unique event names,

Identify the labels and map the index of the list to the structured array: Events.mapID

```
[Events, eventList] = MPfindEventList(Events, doSay);
```

Build the event edges as a structured variable

for each trace, n, and each event, i, with both inclusion and precedence relationships, indicating the "from" and "to" directions as follows:

```
%      Edges(n).inclusion(i).to
%      Edges(n).inclusion(i).from
%      Edges(n).precedence(i).to
%      Edges(n).precedence(i).from
%      Edges(n).named(i).to
%      Edges(n).named(i).from
Edges = MPfindEdges(MP, Events);
```

Derive the cumulative adjacency matrices

```
%      cM.all      : sum of all edges
%      cM.inclusion : sum of inclusion edges
%      cM.precedence : sum of precedence edges
%      cM.named     : sum of named edges

cM = MPmatrix(Edges,eventList);
```

Print the N-squared diagram that consists of the event list and cM

```
MPprintN2(eventList, cM, fout, fcsv);
```

```
N-squared diagram:

      ( 1) ( 2) ( 3) ( 4) ( 5) ( 6) ( 7) ( 8) ( 9) (10) (11) (12) (13) (14) (15) (16) (17) (18)
      ROOT      Leader( 1)  0  0  0  4  8  8  8  0  0  0  0  8  4  4  4  0  0
      ROOT      Subordinates( 2) 0  0  0  0  0  0  0 12  0  0  0  0  0  0  0  0  0
      ROOT      Environment( 3) 0  0  0  0  0  0  0  0  0  4  8  8  0  0  0  0  4  0
COMPOSITE recognizeenvironment( 4) 0  0  0  0  4  0  0  0  5  0  0  0  0  0  0  0  0  1
COMPOSITE receiveinput( 5) 0  0  0  0  0  8  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE decidecorrectly( 6) 0  0  0  0  0  0  8  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE actcorrectly( 7) 0  0  0  0  0  0  0  0  0  0  8  0  0  0  0  0  0  0
COMPOSITE Subordinates( 8) 0  0  0  6  0  0  0  0 10  0  0  0  6  0  0  0  2  0
COMPOSITE communicateobservation( 9) 0  0  0  0  8  0  0  0  0  0  0  0  0  2  0  0  0  0
COMPOSITE notcontainproblem(10) 0  0  0  4  0  0  0  0  0  0  4  0  4  0  0  0  0  0
COMPOSITE desiredresult(11) 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE containproblem(12) 0  0  0  6  0  0  0  0  0  0  4  0 10  0  0  0  4  0
COMPOSITE notrecognizeenvironment(13) 0  0  0  0  4  0  0  0  5  0  0  0  0  4  0  0  0  1
COMPOSITE notreceiveinput(14) 0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  0  0  0
COMPOSITE notdecidecorrectly(15) 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  0  0
COMPOSITE notactcorrectly(16) 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  4  0
COMPOSITE notdesiredresult(17) 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE notcommunicateobservation(18) 0  0  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0
```

Build the DSM.

The adjacency matrix, cM, aligns interactions FROM the row as the "source" and TO the column as the "target." This is known as the input-row (IR) convention. Conversely, the design structure matrix (DSM) uses the input column (IC) convention and so the DSM is a transpose of cM. Also, the DSM does not calculate the accumulated number of interactions, it just indicates that a connection exists, as a one (1) and that if no interface exists, indicated as a "0".

```
DSM = MPdsm(cM.all);
```

Execute DSM Clustering script

```
MPanalyzeDSM;
```

```
Cluster # 1
  Leader( 1)
  recognizeenvironment( 4)
  receiveinput( 5)
  Subordinate( 8)
  communicateobservation( 9)
  containproblem( 12)
  notrecognizeenvironment( 13)
  notreceiveinput( 14)
  notcommunicateobservation( 18)

Cluster # 2
  Subordinates( 2)
  Subordinate( 8)
  notcommunicateobservation( 18)

Cluster # 3
  Environment( 3)
  containproblem( 12)
  notdesiredresult( 17)

Cluster # 4
  notdecidecorrectly( 15)
  notactcorrectly( 16)

Cluster # 5
  notcontainproblem( 10)
  desiredresult( 11)

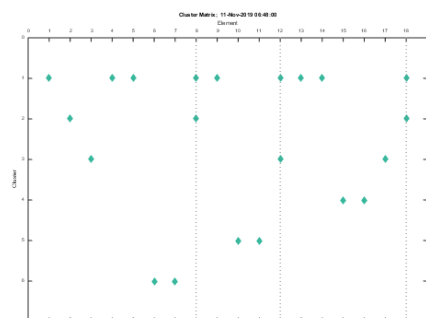
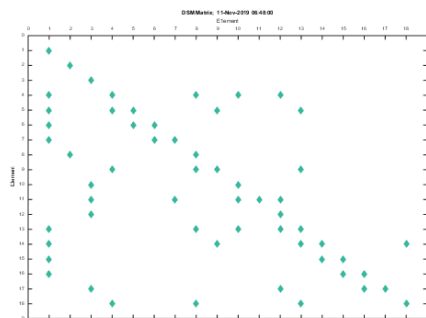
Cluster # 6
  decidecorrectly( 6)
  actcorrectly( 7)
```

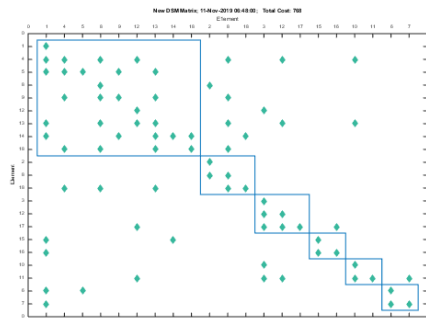
Initial Design Structured Matrix (DSM):

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Leader(1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Subordinates(2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Environment(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
recognizeenvironment(4)	4	0	0	0	0	0	0	0	6	0	4	0	6	0	0	0	0	0
receiveinput(5)	8	0	0	4	0	0	0	0	8	0	0	0	4	0	0	0	0	0
decidecorrectly(6)	8	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0
actcorrectly(7)	8	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
Subordinate(8)	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
communicateobservation(9)	0	0	0	5	0	0	0	10	0	0	0	0	5	0	0	0	0	0
notcontainproblem(10)	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
desiredresult(11)	0	0	8	0	0	0	0	0	0	4	0	4	0	0	0	0	0	0
containproblem(12)	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
notrecognizeenvironment(13)	8	0	0	0	0	0	0	6	0	4	0	10	0	0	0	0	0	0
notreceiveinput(14)	4	0	0	0	0	0	0	0	2	0	0	0	4	0	0	0	0	2
notdecidecorrectly(15)	4	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0
notactcorrectly(16)	4	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0
notdesiredresult(17)	0	0	4	0	0	0	0	0	0	0	0	4	0	0	0	4	0	0
notcommunicateobservation(18)	0	0	0	1	0	0	0	2	0	0	0	0	1	0	0	0	0	0

Updated Design Structured Matrix (DSM):

	(1)	(4)	(5)	(8)	(9)	(12)	(13)	(14)	(18)	(2)	(8)	(18)	(3)	(12)	(17)	(15)	(16)	(10)	(11)	(6)
Leader(1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
recognizeenvironment(4)	4	0	0	6	0	6	0	0	0	0	6	0	0	6	0	0	0	0	4	0
receiveinput(5)	8	4	0	0	8	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0
Subordinate(8)	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0
communicateobservation(9)	0	5	0	10	0	0	5	0	0	0	10	0	0	0	0	0	0	0	0	0
containproblem(12)	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0
notrecognizeenvironment(13)	8	0	0	6	0	10	0	0	0	6	0	0	10	0	0	0	0	4	0	0
notreceiveinput(14)	4	0	0	0	2	0	4	0	2	0	0	2	0	0	0	0	0	0	0	0
notcommunicateobservation(18)	0	1	0	2	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0
Subordinates(2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Subordinate(8)	0	0	0	0	0	0	0	0	0	12	0	0	0	0	0	0	0	0	0	0
notcommunicateobservation(18)	0	1	0	2	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0
Environment(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
containproblem(12)	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0
notdesiredresult(17)	0	0	0	0	0	4	0	0	0	0	0	0	4	4	0	0	4	0	0	0
notdecidecorrectly(15)	4	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
notactcorrectly(16)	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0
notcontainproblem(10)	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0
desiredresult(11)	0	0	0	0	0	4	0	0	0	0	0	0	8	4	0	0	0	4	0	0
decidecorrectly(6)	8	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
actcorrectly(7)	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8





Cluster Member List

11-Nov-2019 06:48:00

Cluster #1

- Leader (1)
- recognizeenvironment (4)
- receiveinput (5)
- Subordinate (8)
- communicateobservation (9)
- containproblem (12)
- notrecognizeenvironment (13)
- notreceiveinput (14)
- notcommunicateobservation (18)

Cluster #2

- Subordinates (2)
- Subordinate (8)
- notcommunicateobservation (18)

Cluster #3

- Environment (3)
- containproblem (12)
- notdesiredresult (17)

Cluster #4

- notdecidecorrectly (15)
- notactcorrectly (16)

Cluster #5

- notcontainproblem (10)
- desiredresult (11)

Cluster #6

- decidecorrectly (6)
- actcorrectly (7)

C.2 Object-to-Effect Model

C.2.1 DSM Derived from the Object-to-Effect Model

This script builds an N-squared diagram, based on the directed graph of Monterey Phoenix Output

Contents

- Script: MPbuildN2.m
- Clear all variables and screens
- Add the script location to the path
- Enter the filename and open the output files
- Open the output files
- Indicate whether to include "Say" events in the N-squared diagram.
- Read the .json file
- Determine the number of traces, events, and links
- Derive the event labels as a structured variable
- Find the list of unique event names,
- Build the event edges as a structured variable
- Derive the cumulative adjacency matrices
- Print the N-squared diagram that consists of the event list and cM
- Build the DSM.
- Execute DSM Clustering script

Called Functions

- MPdsm
- MPfindEdges
- MPfindEventList
- MPfindEvents
- MPmatrix
- MPOpenOutput
- MPprintN2
- MPreadWNGv3

Script: MPbuildN2.m

The script is compatible with MP-Firebird or MP-Gryphon, versions 3.5 and 4. The script operates on MP .wng output of code and graphs.

```
% Written by
% John Quartuccio

% 30 SEP 2016 executable script readWNGv3

% 12 JAN 2018 break-out of sub-functions

% 29 Jul 2018 Updated to v3 of MP

% 08 AUG 2018 Updated to include input and output

% 22 AUG 2018 Updated comments

% 10 NOV 2019 Updated for publication
%
% The script reads the .json file and derives a single list of unique
% events. Then tabulates all event interactions, with event rows as the
% source (from) and event columns as the target (to) of the interactions.
%
% Note: the first few characters may contain an encoding artifact: "i>?"
% These characters need to be deleted in order to read by Matlab.
%
% The convention used has been to open the <filename>.wng file in Matlab,
% and then delete the characters prior to "{"entities":..." and save as
% <filename>.json
%
% The following functions need to exist on the Matlab path, listed here in
% order of occurrence:

%     MPOpenOutput.m      Opens the output files, based on <filename>

%     MPreadWNGv3.m      Reads the .wng file in MP version 3 format

%     MPfindEvents.m      Finds the events

%     MPfindEventList.m   Builds the unique list of events
```

```
% MPfindEdges.m      Finds the edges
% MPmatrix.m         Derives the cumulative matrix of all edges
% MPprintN2.m        Prints the N-squared diagram to .csv
% MPdsm.m            Builds the Design Structure Matrix based on the
%                    N-squared diagram for further cluster analysis
% MPanalyzeDSM.m     Executes the cluster analysis
```

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;
path(oldpath, 'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename and open the output files

```
filename = 'objectToEffectv22scope1111';
% filename = 'objectToEffectv22scope2111';
% filename = 'objectToEffectv22scope2222';
```

Open the output files

```
MPOpenOutput;
```

Indicate whether to include "Say" events in the N-squared diagram.

```
doSay = false;% excludes "Say" events
```

Read the .json file

The parser returns a string variable comprised of the entire .json file. This function populates a structured variable, MP, as follows:

```
% MP.entities.traces.<guid of trace>.guid    <guid of trace>
%                                           .layout    <SEQUENCE>
%                                           <FORCE>
%                                           <SWIMLANE>
%                                           .marked    <logical>
%                                           .nodes     <guid of nodes>
%                                           .links     <guid of links>

% MP.entities.nodes.<guid of node> .guid    <guid of node>
%                                           .name      <user-defined event name>
%                                           .type      <ROOT>
%                                           <COMPOSITE>
%                                           <ATOM>
%                                           <SAY>
%                                           .x         <location>
%                                           .y         <location>
%                                           .collapsed <logical>
%                                           .hidden    <logical>
% MP.entities.links.<guid of link> .guid    <guid of link>
```

```
%
                .type    <IN>
%
                <FOLLOWS>
%
                <NAMED>
%
                .text    <user-defined text>
%
                .source  <guid of source event>
%
                .target  <guid of target event>
%
                .points  <x,y location>
%
                .curviness <value>

MP = MPreadWNGv3(filename);
```

Determine the number of traces, events, and links

```
numTraces = length(struct2cell(MP.entities.traces));
numEvents = length(struct2cell(MP.entities.nodes));
numEdges  = length(struct2cell(MP.entities.links));
```

Derive the event labels as a structured variable

For each trace, n, and each event, i, as follows:

```
%      Events(n).name{i} : the user-defined name of each event
%      Events(n).type{i} : the type follows the following description:
%      <ROOT>, <ATOM>, <COMPOSITE>, <SAY>
%      Events(n).guid(i) : the graph ID for each event in each trace
%      Events(n).ID(i)   : the ID is the reference id for each trace
%      Events(n).mapID(i) : empty field that used to map each event
%                        label to a list of unique names
Events = MPfindEvents(MP);
```

Find the list of unique event names,

Identify the labels and map the index of the list to the structured array: Events.mapID

```
[Events, eventList] = MPfindEventList(Events, doSay);
```

Build the event edges as a structured variable

for each trace, n, and each event, i, with both inclusion and precedence relationships, indicating the "from" and "to" directions as follows:

```
%      Edges(n).inclusion(i).to
%      Edges(n).inclusion(i).from
%      Edges(n).precedence(i).to
%      Edges(n).precedence(i).from
%      Edges(n).named(i).to
%      Edges(n).named(i).from
Edges = MPfindEdges(MP, Events);
```

Derive the cumulative adjacency matrices

```
%      cM.all      : sum of all edges
%      cM.inclusion : sum of inclusion edges
%      cM.precedence : sum of precedence edges
%      cM.named    : sum of named edges
```



```
cM = MPmatrix(Edges,eventList);
```

Print the N-squared diagram that consists of the event list and cM

```
MPprintN2(eventList, cM, fout, fcsv);
```

N-squared diagram:

		(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
ROOT	Environment(1)	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0
ROOT	Objects(2)	0	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0
ROOT	Sensors(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0
ROOT	Processors(4)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18
ROOT	Actors(5)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Environment_Function(6)	0	0	0	0	0	8	9	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_include_problem(7)	0	0	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0	0
COMPOSITE	success(8)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Object(9)	0	0	0	0	0	0	0	0	8	18	0	0	0	0	0	0	0	0
COMPOSITE	is_not_of_interest(10)	0	0	0	0	0	0	0	0	0	8	7	0	0	0	0	0	0	0
COMPOSITE	Object_Function(11)	0	0	0	0	0	0	0	0	0	0	16	17	0	0	0	0	0	0
COMPOSITE	present_observation(12)	0	0	0	0	0	0	0	0	0	0	0	15	0	0	14	0	0	0
COMPOSITE	not_receive_effect(13)	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Sensor(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0
COMPOSITE	Sensor_Function(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	12	0	0
COMPOSITE	detect_object(16)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0
COMPOSITE	send_data(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Processor(18)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Processor_Function(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	receive_data(20)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	merge_data(21)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	interpret_as_not_object(22)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	send_merged_data(23)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Actor(24)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	Actor_Function(25)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	receive_merged_data(26)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_act(27)	0	0	0	0	0	0	0	0	0	0	0	0	16	0	0	0	0	0
COMPOSITE	not_receive_merged_data(28)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_send_merged_data(29)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_merge_data(30)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	interpret_as_unknown(31)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_receive_data(32)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_send_data(33)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_detect_object(34)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	not_present_observation(35)	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0
COMPOSITE	include_problem(36)	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	is_of_interest(37)	0	0	0	0	0	0	0	0	10	9	0	0	0	0	0	0	0	0
COMPOSITE	receive_effect(38)	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	interpret_as_object(39)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE	act(40)	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
COMPOSITE	fail(41)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Build the DSM.

The adjacency matrix, cM, aligns interactions FROM the row as the "source" and TO the column as the "target." This is known as the input-row (IR) convention. Conversely, the design structure matrix (DSM) uses the input column (IC) convention and so the DSM is a transpose of cM. Also, the DSM does not calculate the accumulated number of interactions, it just indicates that a connection exists, as a one (1) and that if no interface exists, indicated as a "0".

```
DSM = MPdsm(cM.all);
```

Execute DSM Clustering script

MPAnalyzeDSM;

Published with MATLAB® R2014b

```
function M = MPdsm(A)
M = A.';
for i = 1 : length(A)
    for j = 1 : length(A)
        if (M(i,j) > 0) M(i,j) = 1; end
        if (i == j) M(i,j) = 1; end
    end
end
```

```

function Edges = MPfindEdges(MP, Events)
% define a structured variable for each trace, n, for the title, type,
% and refID of each event, i, in the trace
%
numTraces = length(struct2cell(MP.entities.traces));
numEdges = length(struct2cell(MP.entities.links));
edgeGuid = char(fieldnames(MP.entities.links));

Edges = struct('inclusion',[],'precedence',[],'named',[]);
Edges.inclusion = struct('to',[],'from',[]);
Edges.precedence = struct('to',[],'from',[]);
Edges.named = struct('to',[],'from',[]);

%
%
% find the guid for each edge, n, and then relate this to the Edges
% structured matrix
%   currentEdge.guid <guid>
%           .type "IN", "FOLLOWS", or "NAMED"
%           .text <user-defined text>
%           .source <guid of source>
%           .target <guid of target>
%   Events      .guid
%
%
iSourceIn = 0; iTargetIn = 0;
iSourcePrec = 0; iTargetPrec = 0;
iSourceNamed = 0; iTargetNamed = 0;
for n = 1 : numEdges
    currentEdge = eval(strcat('MP.entities.links.',edgeGuid(n,:)));
    currentSource = valid_field(currentEdge.source);
    currentTarget = valid_field(currentEdge.target);

    %
    % find the name for the target and source, based on the guid,
    %
    for i = 1 : numTraces
        for j = 1 : length(Events(i).guid)
            %
            if (strcmp(currentSource, Events(i).guid{j}))
                if (strcmp('IN',currentEdge.type))
                    iSourceIn = iSourceIn + 1;
                    Edges.inclusion.from{iSourceIn} = Events(i).mapID{j};
                end % (if IN)

                if (strcmp('FOLLOWS',currentEdge.type))
                    iSourcePrec = iSourcePrec + 1;
                    Edges.precedence.from{iSourcePrec} = Events(i).mapID{j};
                end % (if FOLLOWS)

                if (strcmp('NAMED',currentEdge.type))
                    iSourceNamed = iSourceNamed + 1;
                    Edges.named.from{iSourceNamed} = Events(i).mapID{j};
                end % (if IN)
            end % (if current source equal to guid)
        %
        if (strcmp(currentTarget, Events(i).guid{j}))
            if (strcmp('IN',currentEdge.type))
                iTargetIn = iTargetIn + 1;
                Edges.inclusion.to{iTargetIn} = Events(i).mapID{j};
            end % (if IN)

            if (strcmp('FOLLOWS',currentEdge.type))
                iTargetPrec = iTargetPrec + 1;
                Edges.precedence.to{iTargetPrec} = Events(i).mapID{j};
            end % (if FOLLOWS)

            if (strcmp('NAMED',currentEdge.type))
                iTargetNamed = iTargetNamed + 1;
                Edges.named.to{iTargetNamed} = Events(i).mapID{j};
            end % (if NAMED)
        end % (if current source equal to guid)

        end % (Events within the trace)
    end % (number of traces)
end

```

```

end % end of for loop (numEdges)

end % end of function (MPfindEdges)

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
if ~isletter(str(1))
    str = ['alpha_' str];
end
str(~isletter(str) & ~('0' <= str & str <= '9')) = '_';
end

```

Published with MATLAB® R2014b

```

function [Events, eventList] = MPfindEventList(Events, doSay)
% MPbuildUnique finds the unique list of event names and provide a
% reference to the index of the unique list.
%
%
% find the number of traces, based on the Events structured array
numTraces = length(Events);

%
% for each type of event, (root, composite, atom, or say), find the unique
% names used
uniqueEvents = struct('root',[], 'composite',[], 'atom',[], 'say',[]);
ir = 0; ic = 0; ia = 0; isay = 0;
for n = 1:numTraces
    for i = 1: length(Events(n).name)
        if (strcmp(Events(n).type(i), 'ROOT'))
            if(ir == 1)
                ir = ir + 1;
                uniqueEvents.root{ir} = Events(n).name{i};
            else
                flag = 0;
                for j = 1:ir
                    if (strcmp(uniqueEvents.root{j},Events(n).name{i}))
                        flag = 0;
                    else
                        flag = flag + 1;
                    end
                end
                if (flag == ir)
                    ir = ir + 1;
                    uniqueEvents.root{ir} = Events(n).name{i};
                end
            end
        elseif(strcmp(Events(n).type(i), 'COMPOSITE'))
            if(ir == 1)
                ic = ic + 1;
                uniqueEvents.composite{ic} = Events(n).name{i};
            else
                flag = 0;
                for j = 1:ic
                    if (strcmp(uniqueEvents.composite{j},Events(n).name{i}))
                        flag = 0;
                    else
                        flag = flag + 1;
                    end
                end
                if (flag == ic)
                    ic = ic + 1;
                    uniqueEvents.composite{ic} = Events(n).name{i};
                end
            end
        end
    end
end

```

```

elseif(strcmp(Events(n).type(i), 'ATOM'))
    if(ia == 1)
        ia = ia + 1;
        uniqueEvents.atom{ia} = Events(n).name{i};
    else
        flag = 0;
        for j = 1:ia
            if (strcmp(uniqueEvents.atom{j},Events(n).name(i)))
                flag = 0;
            else
                flag = flag + 1;
            end
        end
        if (flag == ia)
            ia = ia + 1;
            uniqueEvents.atom{ia} = Events(n).name{i};
        end
    end
end

elseif(strcmp(Events(n).type(i), 'SAY'))
    if(isay == 1)
        isay = isay + 1;
        uniqueEvents.say{isay} = Events(n).name{i};
    else
        flag = 0;
        for j = 1:isay
            if (strcmp(uniqueEvents.say{j},Events(n).name(i)))
                flag = 0;
            else
                flag = flag + 1;
            end
        end
        if (flag == isay)
            isay = isay + 1;
            uniqueEvents.say{isay} = Events(n).name{i};
        end
    end
end
end
end
end
itotal = ir + ic + ia + isay;

eventList = struct('name',[],'type',[]);
i = 0;
for j = 1:length(uniqueEvents.root)
    i = i + 1;
    eventList.name{i} = uniqueEvents.root{j};
    eventList.type{i} = 'ROOT';
end

for j = 1:length(uniqueEvents.composite)
    i = i + 1;
    eventList.name{i} = uniqueEvents.composite{j};
    eventList.type{i} = 'COMPOSITE';
end

for j = 1:length(uniqueEvents.atom)
    i = i + 1;
    eventList.name{i} = uniqueEvents.atom{j};
    eventList.type{i} = 'ATOM';
end

if (doSay)
    for j = 1:length(uniqueEvents.say)
        i = i + 1;
        eventList.name{i} = uniqueEvents.say{j};
        eventList.type{i} = 'SAY';
    end
end

for n = 1:numTraces
    for i = 1:length(Events(n).name)
        for j = 1:length(eventList.name)
            if strcmp(Events(n).name{i}, eventList.name{j})
                Events(n).mapID{i} = j;
            end
        end
    end
end
end
end
end

```

```

function Events = MPfindEvents(MP)
% define a structured variable for the events in each trace, n, including
% the guid, name, type, and index of each event, i, in the trace
%
%
% determine the number of traces
numTraces = length(struct2cell(MP.entities.traces));
traceGuid = char(fieldnames(MP.entities.traces));

%
%
% define the Labels variable
Events = struct('name',[],'type',[],'guid',[],'ID',[],'mapID',[]);

%
%
% find the guid for each node, i, in each trace, n and then relate this to
% the Events structured matrix
%
for n = 1 : numTraces
    currentTrace = eval(strcat('MP.entities.traces.',traceGuid(n,:)));
    numEventsInTrace = length(currentTrace.nodes);
    for i = 1 : numEventsInTrace
        c = valid_field(currentTrace.nodes{i});
        Events(n).name(i) = eval(strcat('MP.entities.nodes.',c,'.name'));
        Events(n).type(i) = eval(strcat('MP.entities.nodes.',c,'.type'));
        Events(n).guid(i) = c;
        Events(n).ID(i) = i;
    end % end of for loop (numEventsInTrace)
end % end of for loop (numTraces)

end % end of function (MPfindEvents)

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
if ~isletter(str(1))
    str = ['alpha_' str];
end
str(~isletter(str) & ~('0' <= str & str <= '9')) = '_';
end

```

```

function cM = MPmatrix(Edges,eventList)
% John Quartuccio
% 4 Jan 2018
% revised 7 Aug 2018
% Function MPmatrix(Edges,eventList) returns the cumulative adjacency
% matrix for inclusion, precedence, and named edges.
%
% Edges.<type>.<from/to> captures the reference ID consistent with the
% order of the index of the eventList
%
%
% Initialize the cumulative adjacency matrices
cM.all = zeros(length(eventList.name),length(eventList.name));
cM.inclusion = cM.all; cM.precedence = cM.all; cM.named = cM.all;

for i = 1 : length(Edges.inclusion.from)
    cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) = ...
        cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) + 1;
end % (end of for loop of inclusion edges)

```

```

for i = 1 : length(Edges.precedence.from)
    cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) = ...
        cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) + 1;
end % (end of for loop of precedence edges)

for i = 1 : length(Edges.named.from)
    cM.named(Edges.named.from{i},Edges.named.to{i}) = ...
        cM.named(Edges.named.from{i},Edges.named.to{i}) + 1;
end % (end of for loop of named edges)

cM.all = cM.inclusion + cM.precedence + cM.named;

end %(end of MPmatrix function)

```

Published with MATLAB® R2014b

```

%
% open the output files
fout = fopen(strcat(filename, '.txt'),'w');
fcsv = fopen(strcat(filename, '.csv'),'w');
ftex = fopen(strcat(filename, '.tex'),'w');

```

Published with MATLAB® R2014b

This function prints the N2 diagram to the screen and output files

```

function MPprintN2(eventList, cM, fout, fcsv)
%
% Print the cumulative matrix, cMa
%
% for the screen output and the text files, determine the width of the
% column on the basis of the length of event names.
    len = longestName(eventList.name);
    % if (len>16), len = 16; end;
    sep = 6;
    colstringformat = strcat('%',num2str(len),'s');
    colnumformat = strcat('%',num2str(sep),'d');
    colindexformat = strcat('%',num2str(sep-4),'s(%2s)');
    rowstringformat = strcat('%',num2str(len),'s(%2s)');

    skip = ' ';
    %
    fprintf ( '\n N-squared diagram:\n');
    fprintf (fout,'\n N-squared diagram:\n');

% Print the column headers
%
    fprintf( '%16s ', ' ');
    fprintf( colstringformat, ' ');
    fprintf(fout, '%16s ', ' ');
    fprintf(fout, colstringformat, ' ');
    fprintf(fcsv, '%s, %s, ', 'Event type', 'Event name', 'index');
%
    for i = 1:length(eventList.name)
        fprintf( colstringformat, eventList.name{i});
        fprintf(fout, colstringformat, eventList.name{i});
        fprintf(fcsv, '%s,', eventList.name{i});
    end % ( end of print column headers)
%
    fprintf( '\n');
    fprintf(fout,'\n');
    fprintf(fcsv,'\n');

    fprintf( strcat('%14s %',num2str(len),'s '), skip, skip);
    fprintf(fout, strcat('%16s %',num2str(len),'s '), skip, skip);
    fprintf(fcsv, '\n , , ');
%
    for i = 1:length(eventList.name)
        fprintf( colindexformat, skip, num2str(i));
        fprintf(fout, colindexformat, skip, num2str(i));
        fprintf(fcsv, ', (%d)', i);
    end % ( end of print column index)

```

```

    fprintf(' \n');
    fprintf(fout, '\n');
    fprintf(fcsv, '\n');

%
% Print out the type, name, and values of the N-squared diagram
% m: index for each row of the cumulative matrix
% n: index for the column of the cumulative matrix
for m = 1 : length(eventList.name);
    fprintf(' %10s ', eventList.type{m});
    fprintf(rowstringformat, eventList.name{m}, num2str(m));
    fprintf(fout, ' %10s ', eventList.type{m});
    fprintf(fout, rowstringformat, eventList.name{m}, num2str(m));
    fprintf(fcsv, '%s,%s,(%s)', ...
        eventList.type{m}, eventList.name{m}, num2str(m));
%
for n = 1 : length(eventList.name);
    fprintf(colnumformat, cM.all(m,n));
    fprintf(fout, colnumformat, cM.all(m,n));
    if (cM.all(m,n) == 0)
        fprintf(fcsv, '%s', skip);
    else
        fprintf(fcsv, '%d', cM.all(m,n));
    end % (end of if)
end % (end of print each column of current row)
fprintf('\n'); fprintf(fout, '\n'); fprintf(fcsv, '\n');
end % (end of print rows)
end % (end of function)

function len = longestName(name)
% returns the longest name of the structured matrix, name.
len = 4;
for i = 1 : length(name)
    x = length(name{i});
    if (x > len)
        len = x;
    end % (end of if)
end % (end of for)
end % (end of function)

```

Published with MATLAB® R2014b

```

function MP = MPreadWNGv3(filename)
% John Quartuccio
% 30 Sep 2016
% updated 24 Apr 2017
% updated 04 Jan 2018
% updated 28 Jul 2018 to MP version 3, output by Gryphon and Firebird
%
% This script reads the .wng file in .json format as produced as file by
% executing Monterey Phoenix through MP-Firebird Analyzer at
% nps.firebird.edu
%
% The routine employs the parse_json.m function,
% downloaded 30 Sep 2016 from
% https://www.mathworks.com/matlabcentral/fileexchange/
% 23393--another--json-parser
% Other parsers available at https://json.org/
%
%
fjson = fopen(strcat(filename, '.json'));
MP = parse_json(fscanf(fjson, '%s'));
fclose(fjson);

```

Published with MATLAB® R2014b

C.2.2 Comparison of Object-to-Effect Models at Scope (1111) and (2111)

This script compares DSM from two sets of MP output files

Contents

- Script: RunCompareDSM
- Clear all variables and screens
- Add the script location to the path
- Enter the filename of the first file
- Build the N2 Diagram for the first file
- Enter the filename of the second file
- Build the N2 Diagram for the second file
- Compare the DSMs

Called Functions

- MPbuildN2
- MPdsm
- MPfindEdges
- MPfindEvents
- MPmatrix
- MPOpenOutput
- MPprintN2
- MPreadWNGv3

Script: RunCompareDSM

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;  
path(oldpath, 'c:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename of the first file

```
filename = 'objectToEffectv22scope1111';  
filenameA = filename;
```

Build the N2 Diagram for the first file

```
MPbuildN2;  
DSMa = MPdsm(cM.all);
```

N-squared diagram:

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)
ROOT Environment(1)	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
ROOT Objects(2)	0	0	0	0	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0
ROOT Sensors(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0	0
ROOT Processors(4)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0
ROOT Actors(5)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE Environment_Function(6)	0	0	0	0	0	0	8	9	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE not_include_problem(7)	0	0	0	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0	0
COMPOSITE success(8)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE Object(9)	0	0	0	0	0	0	0	0	0	8	18	0	0	0	0	0	0	0	0
COMPOSITE is_not_of_interest(10)	0	0	0	0	0	0	0	0	0	0	8	7	0	0	0	0	0	0	0
COMPOSITE Object_Function(11)	0	0	0	0	0	0	0	0	0	0	0	16	17	0	0	0	0	0	0
COMPOSITE present_observation(12)	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	14	0	0
COMPOSITE not_receive_effect(13)	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE Sensor(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0	0
COMPOSITE Sensor_Function(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14	12	0	0	0
COMPOSITE detect_object(16)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	0	0	0
COMPOSITE send_data(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE Processor(18)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18
COMPOSITE Processor_Function(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE receive_data(20)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE merge_data(21)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE interpret_as_not_object(22)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE send_merged_data(23)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COMPOSITE Actor(24)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0


```

COMPOSITE      Actor_Function(25)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      receive_merged_data(26)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_act(27)  0  0  0  0  0  0  0  0  0  0  0  0  16  0  0  0  0  0
COMPOSITE      not_receive_merged_data(28)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_send_merged_data(29)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_merge_data(30)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      interpret_as_unknown(31)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_receive_data(32)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_send_data(33)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_detect_object(34)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      not_present_observation(35)  0  0  0  0  0  0  0  0  0  0  0  2  0  0  0  0  0
COMPOSITE      include_problem(36)  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
COMPOSITE      is_of_interest(37)  0  0  0  0  0  0  0  0  0  10  9  0  0  0  0  0  0  0
COMPOSITE      receive_effect(38)  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  0
COMPOSITE      interpret_as_object(39)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE      act(40)  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  0  0
COMPOSITE      fail(41)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

```

Enter the filename of the second file

```

filename = 'objectToEffectv22scope2111';
filenameB = filename;

```

Build the N2 Diagram for the second file

read the file;

```

MPOpenOutput;
MP = MPreadNVGv3(filename);
% Determine the number of traces, events, and links
numTraces = length(struct2cell(MP.entities.traces));
numEvents = length(struct2cell(MP.entities.nodes));
numEdges = length(struct2cell(MP.entities.links));
% Find the events
Events = MPfindEvents(MP);
% Map events to the event list
for n = 1:numTraces
    for i = 1:length(Events(n).name)
        for j = 1:length(eventList.name)
            if strcmp(Events(n).name{i}, eventList.name{j})
                Events(n).mapID{i} = j;
            end
        end
    end
end
% Build the event edges as a structured variable
Edges = MPfindEdges(MP, Events);
% Derive the cumulative adjacency matrices
cM = MPmatrix(Edges, eventList);
% Print the N-squared diagram that consists of the event list and cM
MPprintN2(eventList, cM, fout, fcsv);
% Find the DSM for the second set
DSMb = MPdsm(cM.all);

```

```

N-squared diagram:
      ( 1) ( 2) ( 3) ( 4) ( 5) ( 6) ( 7) ( 8) ( 9) (10) (11) (12) (13) (14) (15) (16) (17) (18) (19)
ROOT      Environment( 1)  0  0  0  0  0  72  0  0  0  0  0  0  0  0  0  0  0  0  0
ROOT      Objects( 2)  0  0  0  0  0  0  0  0  0  144  0  0  0  0  0  0  0  0  0
ROOT      Sensors( 3)  0  0  0  0  0  0  0  0  0  0  0  0  0  72  0  0  0  0  0
ROOT      Processors( 4)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  72  0
ROOT      Actors( 5)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Environment_Function( 6)  0  0  0  0  0  0  16  18  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  not_include_problem( 7)  0  0  0  0  0  0  0  16  0  32  0  0  0  0  0  0  0  0  0
COMPOSITE  success( 8)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Object( 9)  0  0  0  0  0  0  0  0  0  68  144  0  0  0  0  0  0  0  0
COMPOSITE  is_not_of_interest(10)  0  0  0  0  0  0  0  0  0  0  68  46  0  0  0  0  0  0  0
COMPOSITE  Object_Function(11)  0  0  0  0  0  0  0  0  0  0  0  100  140  0  0  0  0  0
COMPOSITE  present_observation(12)  0  0  0  0  0  0  0  0  0  0  0  0  96  0  0  56  0  0
COMPOSITE  not_receive_effect(13)  0  0  0  0  0  0  0  34  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Sensor(14)  0  0  0  0  0  0  0  0  0  0  0  0  0  72  0  0  0  0  0
COMPOSITE  Sensor_Function(15)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  56  48  0  0
COMPOSITE  detect_object(16)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  48  0
COMPOSITE  send_data(17)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Processor(18)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  72
COMPOSITE  Processor_Function(19)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  receive_data(20)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  merge_data(21)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  interpret_as_not_object(22)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  send_merged_data(23)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Actor(24)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
COMPOSITE  Actor_Function(25)  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0

```

Compare the DSMs

```
Sum of DSM edges from objectToEffectv22scope1111 is 123
Sum of DSM edges from objectToEffectv22scope2111 is 125
i = 10, j = 36: objectToEffectv22scope2111 has relationship from include_problem to is_not_of_interest not shared by objectToEffectv22scope1111
i = 41, j = 36: objectToEffectv22scope2111 has relationship from receive_effect to fail not shared by objectToEffectv22scope1111
```

This script builds an N-squared diagram, based on the directed graph of Monterey Phoenix Output

Contents

- 376

Script: MPbuildN2.m

The script is compatible with MP-Firebird or MP-Gryphon, versions 3.5 and 4. The script operates on MP .wng output of code and graphs.

```
% Written by
% John Quartuccio

% 30 SEP 2016 executable script readWNGv3

% 12 JAN 2018 break-out of sub-functions

% 29 Jul 2018 Updated to v3 of MP

% 08 AUG 2018 Updated to include input and output

% 22 AUG 2018 Updated comments

% 10 NOV 2019 Updated for publication
%
% The script reads the .json file and derives a single list of unique
% events. Then tabulates all event interactions, with event rows as the
% source (from) and event columns as the target (to) of the interactions.
%
% Note: the first few characters may contain an encoding artifact: "ï»¿"
% These characters need to be deleted in order to read by Matlab.
%
% The convention used has been to open the <filename>.wng file in Matlab,
% and then delete the characters prior to "{"entities":..." and save as
% <filename>.json
%
% The following functions need to exist on the Matlab path, listed here in
% order of occurrence:

%   MPOpenOutput.m       Opens the output files, based on <filename>
%   MPreadWNGv3.m        Reads the .wng file in MP version 3 format
%   MPfindEvents.m        Finds the events
%   MPfindEventList.m     Builds the unique list of events
%   MPfindEdges.m         Finds the edges
%   MPmatrix.m            Derives the cumulative matrix of all edges
%   MPprintN2.m           Prints the N-squared diagram to .csv
%   MPdsm.m              Builds the Design Structure Matrix based on the
%                        N-squared diagram for further cluster analysis
%   MPanalyzeDSM.m        Executes the cluster analysis
```

Clear all variables and screens

```
clc; clear all; close all;
```

Add the script location to the path

```
oldpath = path;
path(oldpath, 'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\MPscripts');
```

Enter the filename and open the output files

```
filename = 'objectToEffectv22scope1111'; filename = 'objectToEffectv22scope2111'; filename = 'objectToEffectv22scope2222';
```

Open the output files

```
MPOpenOutput;
```

Indicate whether to include "Say" events in the N-squared diagram.

```
doSay = false;% excludes "Say" events
```

Read the .json file

The parser returns a string variable comprised of the entire .json file. This function populates a structured variable, MP, as follows:

```
% MP.entities.traces.<guid of trace>.guid    <guid of trace>
```

```

%                               .layout    <SEQUENCE>
%                               <FORCE>
%                               <SWIMLANE>
%                               .marked    <logical>
%                               .nodes     <guid of nodes>
%                               .links     <guid of links>

% MP.entities.nodes.<guid of node> .guid    <guid of node>
%                               .name     <user-defined event name>
%                               .type     <ROOT>
%                               <COMPOSITE>
%                               <ATOM>
%                               <SAY>
%                               .x        <location>
%                               .y        <location>
%                               .collapsed <logical>
%                               .hidden   <logical>
% MP.entities.links.<guid of link> .guid    <guid of link>
%                               .type     <IN>
%                               <FOLLOWS>
%                               <NAMED>
%                               .text     <user-defined text>
%                               .source   <guid of source event>
%                               .target   <guid of target event>
%                               .points   <x,y location>
%                               .curviness <value>

MP = MPreadWNGv3(filename);

```

Determine the number of traces, events, and links

```

numTraces = length(struct2cell(MP.entities.traces));
numEvents = length(struct2cell(MP.entities.nodes));
numEdges  = length(struct2cell(MP.entities.links));

```

Derive the event labels as a structured variable

For each trace, n, and each event, i, as follows:

```

%      Events(n).name{i} : the user-defined name of each event
%      Events(n).type{i} : the type follows the following description:
%                          <ROOT>, <ATOM>, <COMPOSITE>, <SAY>
%      Events(n).guid{i} : the graph ID for each event in each trace
%      Events(n).ID{i}   : the ID is the reference id for each trace
%      Events(n).mapID{i} : empty field that used to map each event
%                          label to a list of unique names
Events = MPFindEvents(MP);

```

Find the list of unique event names,

Identify the labels and map the index of the list to the structured array: Events.mapID

```

[Events, eventList] = MPFindEventList(Events, doSay);

```

Build the event edges as a structured variable

for each trace, n, and each event, i, with both inclusion and precedence relationships, indicating the "from" and "to" directions as follows:

```
%      Edges(n).inclusion(i).to
%      Edges(n).inclusion(i).from
%      Edges(n).precedence(i).to
%      Edges(n).precedence(i).from
%      Edges(n).named(i).to
%      Edges(n).named(i).from
Edges = MPfindEdges(MP, Events);
```

Derive the cumulative adjacency matrices

```
%      cM.all      : sum of all edges
%      cM.inclusion  : sum of inclusion edges
%      cM.precedence : sum of precedence edges
%      cM.named     : sum of named edges
cM = MPmatrix(Edges,eventList);
```

Print the N-squared diagram that consists of the event list and cM

```
MPprintN2(eventList, cM, fout, fcsv);
```

Build the DSM.

The adjacency matrix, cM, aligns interactions FROM the row as the "source" and TO the column as the "target." This is known as the input-row (IR) convention. Conversely, the design structure matrix (DSM) uses the input column (IC) convention and so the DSM is a transpose of cM. Also, the DSM does not calculate the accumulated number of interactions, it just indicates that a connection exists, as a one (1) and that if no interface exists, indicated as a "0".

```
DSM = MPdsm(cM.all);
```

Execute DSM Clustering script

```
MPanalyzeDSM;
```

Published with MATLAB® R2014b

```
function M = MPdsm(A)
M = A.';
for i = 1 : length(A)
    for j = 1 : length(A)
        if (M(i,j) > 0) M(i,j) = 1; end
        if (i == j) M(i,j) = 1; end
    end
end
```

Published with MATLAB® R2014b

```
function Edges = MPfindEdges(MP, Events)
% define a structured variable for each trace, n, for the title, type,
% and refID of each event, i, in the trace
%
numTraces = length(struct2cell(MP.entities.traces));
numEdges = length(struct2cell(MP.entities.links));
edgeGuid = char(fieldnames(MP.entities.links));
```

```

Edges = struct('inclusion',[],'precedence',[],'named',[]);
Edges.inclusion = struct('to',[],'from',[]);
Edges.precedence = struct('to',[],'from',[]);
Edges.named = struct('to',[],'from',[]);

%
%
% find the guid for each edge, n, and then relate this to the Edges
% structured matrix
%   currentEdge.guid <guid>
%   .type "IN", "FOLLOWS", or "NAMED"
%   .text <user-defined text>
%   .source <guid of source>
%   .target <guid of target>
%   Events .guid
%
iSourceIn = 0; iTargetIn = 0;
iSourcePrec = 0; iTargetPrec = 0;
iSourceNamed = 0; iTargetNamed = 0;
for n = 1 : numEdges
    currentEdge = eval(strcat('MP.entities.links.',edgeGuid(n,:)));
    currentSource = valid_field(currentEdge.source);
    currentTarget = valid_field(currentEdge.target);

%
%
% find the name for the target and source, based on the guid,
%
for i = 1 : numTraces
    for j = 1 : length(Events(i).guid)
%
        if (strcmp(currentSource, Events(i).guid{j}))
            if (strcmp('IN',currentEdge.type))
                iSourceIn = iSourceIn + 1;
                Edges.inclusion.from{iSourceIn} = Events(i).mapID{j};
            end % (if IN)

            if (strcmp('FOLLOWS',currentEdge.type))
                iSourcePrec = iSourcePrec + 1;
                Edges.precedence.from{iSourcePrec} = Events(i).mapID{j};
            end % (if FOLLOWS)

            if (strcmp('NAMED',currentEdge.type))
                iSourceNamed = iSourceNamed + 1;
                Edges.named.from{iSourceNamed} = Events(i).mapID{j};
            end % (if IN)
        end % (if current source equal to guid)

%
        if (strcmp(currentTarget, Events(i).guid{j}))
            if (strcmp('IN',currentEdge.type))
                iTargetIn = iTargetIn + 1;
                Edges.inclusion.to{iTargetIn} = Events(i).mapID{j};
            end % (if IN)

            if (strcmp('FOLLOWS',currentEdge.type))
                iTargetPrec = iTargetPrec + 1;
                Edges.precedence.to{iTargetPrec} = Events(i).mapID{j};
            end % (if FOLLOWS)

            if (strcmp('NAMED',currentEdge.type))
                iTargetNamed = iTargetNamed + 1;
                Edges.named.to{iTargetNamed} = Events(i).mapID{j};
            end % (if NAMED)
        end % (if current source equal to guid)

        end % (Events within the trace)
    end % (number of traces)

end % end of for loop (numEdges)

end % end of function (MPfindEdges)

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
    if ~isletter(str(1))
        str = ['alpha_' str];
    end
end

```

```

        end
        str(~isletter(str) & ~('0' <= str & str <= '9')) = '_';
    end
end

```

Published with MATLAB® R2014b

```

function Events = MPfindEvents(MP)
% define a structured variable for the events in each trace, n, including
% the guid, name, type, and index of each event, i, in the trace
%
%
% determine the number of traces
numTraces = length(struct2cell(MP.entities.traces));
traceGuid = char(fieldnames(MP.entities.traces));

%
%
% define the Labels variable
%
Events = struct('name',[],'type',[],'guid',[],'ID',[],'mapID',[]);

%
%
% find the guid for each node, i, in each trace, n and then relate this to
% the Events structured matrix
%
for n = 1 : numTraces
    currentTrace = eval(strcat('MP.entities.traces.',traceGuid(n,:)));
    numEventsInTrace = length(currentTrace.nodes);
    for i = 1 : numEventsInTrace
        c = valid_field(currentTrace.nodes{i});
        Events(n).name{i} = eval(strcat('MP.entities.nodes.',c,'.name'));
        Events(n).type{i} = eval(strcat('MP.entities.nodes.',c,'.type'));
        Events(n).guid{i} = c;
        Events(n).ID{i} = i;
    end % end of for loop (numEventsInTrace)
end % end of for loop (numTraces)

end % end of function (MPfindEvents)

```

```

function str = valid_field(str)
% Function copied from parse_json, used here so that structured
% variables in guid are legal in MATLAB.
%
% From MATLAB doc: field names must begin with a letter, which may be
% followed by any combination of letters, digits, and underscores.
% Invalid characters will be converted to underscores, and the prefix
% "alpha_" will be added if first character is not a letter.
if ~isletter(str(1))
    str = ['alpha_' str];
end
str(~isletter(str) & ~('0' <= str & str <= '9')) = '_';
end

```

Published with MATLAB® R2014b

```

function cM = MPmatrix(Edges,eventList)
% John Quantuccio
% 4 Jan 2018
% revised 7 Aug 2018
% Function MPmatrix(Edges,eventList) returns the cumulative adjacency
% matrix for inclusion, precedence, and named edges.
%
% Edges.<type>.<from/to> captures the reference ID consistent with the
% order of the index of the eventList
%
% Initialize the cumulative adjacency matrices
cM.all = zeros(length(eventList.name),length(eventList.name));
cM.inclusion = cM.all; cM.precedence = cM.all; cM.named = cM.all;

for i = 1 : length(Edges.inclusion.from)
    cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) = ...

```

```

        cM.inclusion(Edges.inclusion.from{i},Edges.inclusion.to{i}) + 1;
    end % (end of for loop of inclusion edges)

    for i = 1 : length(Edges.precedence.from)
        cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) = ...
            cM.precedence(Edges.precedence.from{i},Edges.precedence.to{i}) + 1;
    end % (end of for loop of precedence edges)

    for i = 1 : length(Edges.named.from)
        cM.named(Edges.named.from{i},Edges.named.to{i}) = ...
            cM.named(Edges.named.from{i},Edges.named.to{i}) + 1;
    end % (end of for loop of named edges)

    cM.all = cM.inclusion + cM.precedence + cM.named;

end % (end of MPmatrix function)

```

Published with MATLAB® R2014b

```

%
% open the output files
fout = fopen(strcat(filename, '.txt'), 'w+');
fcsv = fopen(strcat(filename, '.csv'), 'w+');
ftex = fopen(strcat(filename, '.tex'), 'w+');

```

Published with MATLAB® R2014b

This function prints the N2 diagram to the screen and output files

```

function MPprintN2(eventList, cM, fout, fcsv)
%
% Print the cumulative matrix, cMa
%
% For the screen output and the text files, determine the width of the
% column on the basis of the length of event names.
    len = longestName(eventList.name);
    if (len>16), len = 16; end;
    sep = 6;
    colstringformat = strcat('%',num2str(len),'s');
    colnumformat = strcat('%',num2str(sep),'d');
    colindexformat = strcat('%',num2str(sep-4),'s(%2s)');
    rowstringformat = strcat('%',num2str(len),'s(%2s)');

    skip = ' ';
    %
    fprintf ( '\n N-squared diagram:\n');
    fprintf (fout, '\n N-squared diagram:\n');
%
% Print the column headers
%
    fprintf( '%16s ', ' ');
    fprintf( colstringformat, ' ');
    fprintf(fout, '%16s ', ' ');
    fprintf(fout, colstringformat, ' ');
    fprintf(fcsv, '%s, %s, %s, 'Event type', 'Event name', 'index');
%
    for i = 1:length(eventList.name)
%
        fprintf( colstringformat, eventList.name{i});
        fprintf(fout, colstringformat, eventList.name{i});
        fprintf(fcsv, '%s,', eventList.name{i});
    end % ( end of print column headers)
%
    fprintf( '\n');
    fprintf(fout, '\n');
    fprintf(fcsv, '\n');

    fprintf( strcat('%14s %',num2str(len),'s '), skip, skip);
    fprintf(fout, strcat('%16s %',num2str(len),'s '), skip, skip);
    fprintf(fcsv, '\n , , ');
%
    for i = 1:length(eventList.name)
        fprintf( colindexformat, skip, num2str(i));
        fprintf(fout, colindexformat, skip, num2str(i));
        fprintf(fcsv, ', (%d)', 1);
    end
end

```



```

end % ( end of print column index)
%
fprintf( '\n');
fprintf(fout, '\n');
fprintf(fcsv, '\n');

%
% Print out the type, name, and values of the N-squared diagram
% m: index for each row of the cumulative matrix
% n: index for the column of the cumulative matrix
for m = 1 : length(eventList.name);
    fprintf( '%10s ', eventList.type{m});
    fprintf( rowstringformat, eventList.name{m}, num2str(m));
    fprintf(fout, '%10s ', eventList.type{m});
    fprintf(fout, rowstringformat, eventList.name{m}, num2str(m));
    fprintf(fcsv, '%s,%s,%s)', ...
        eventList.type{m}, eventList.name{m}, num2str(m));
%
for n = 1 : length(eventList.name);
    fprintf( columnformat, cM.all(m,n));
    fprintf(fout, columnformat, cM.all(m,n));
    if (cM.all(m,n) == 0)
        fprintf(fcsv, '%s', skip);
    else
        fprintf(fcsv, '%d', cM.all(m,n));
    end % (end of if)
end % (end of print each column of current row)
fprintf('\n'); fprintf(fout, '\n'); fprintf(fcsv, '\n');
end % (end of print rows)
end % (end of function)

function len = longestName(name)
% returns the longest name of the structured matrix, name.
len = 4;
for i = 1 : length(name)
    x = length(name{i});
    if (x > len)
        len = x;
    end % (end of if)
end % (end of for)
end % (end of function)

```

Published with MATLAB® R2014b

```

function MP = MPreadNGv3(filename)
% John Quartuccio
% 30 Sep 2016
% updated 24 Apr 2017
% updated 04 Jan 2018
% updated 28 Jul 2018 to MP version 3, output by Gryphon and Firebird
%
% This script reads the .wmg file in .json format as produced as file by
% executing Monterey Phoenix through MP-Firebird Analyzer at
% nps.firebird.edu
%
% The routine employs the parse_json.m function,
% downloaded 30 Sep 2016 from
% https://www.mathworks.com/matlabcentral/fileexchange/
% 23393--another--json-parser
% Other parsers available at https://json.org/
%
%
%
fjson = fopen(strcat(filename, '.json'));
MP = parse_json(fscanf(fjson, '%s'));
fclose(fjson);

```

Published with MATLAB® R2014b

C.2.3 Cluster Analysis of the Object-to-Effect Model

This script executes a cluster algorithm, operating on a Design Structured Matrix (DSM), as derived by Thebeau (2000), cited below.

Contents

- Script MPanalyzeDSM.m
- Add the path containing the clustering functions and scripts
- Builds the DSM.
- Remove events that have no interactions
- Prepare the the DSM for analysis, retaining the original file
- Find the event labels for the DSM analysis, excluding the events removed
- Set-up the clustering parameters
- Run clustering
- Display graphs
- Print out the DSM matrices before and after clustering

Script MPanalyzeDSM.m

```
% Written by
% John Quartuccio

% 12 JAN 2018 employed clustering algorithm to MP

% 08 AUG 2018 revised algorithm to segment into sub-functions

% 25 AUG 2018 added comments

% 10 Nov 2019 edited for publication
%
% The MPanalyzeDSM.m script relies on first executing script MPbuildN2.m
% in order to derive the cM structured matrix, based on the MP output file.
%
% MPanalyzeDSM.m employs the following functions:

% MPdsm.m           Derives the DSM, based on the cM structured matrix

% MPremoveEvents.m  Remove specific events that are not of interest

% MPeventsDSM.m     Aligns event labels with DSM structure

% MPsetupDSM.m      Sets cluster parameters, based on Thebeau

% Cluster.m         Core algorithm created by Thebeau

% MPgraphDSM.m      Graphs the results, based on Thebeau

% MPprintDSM.m      Prints the results, based on Thebeau
%
% The cluster analysis algorithm was downloaded on 06 JAN 2018, from
% http://www.dsmweb.org/?id=121
% citation is as follows:

% Created by: Ronnie E. Thebeau

% System Design and Management Program

% Massachusetts Institute of Technology

% Date: December 2000

% Associated master's thesis available from at
% https://dspace.mit.edu/handle/1721.1/29168
```

Add the path containing the clustering functions and scripts

```
oldpath = path;
path(oldpath,'C:\Users\me\Documents\nps\dissertation\m\MPtoDSM\DSM_MatLab');
print_flag = 0;

% The adjacency matrix, cM, aligns interactions FROM the row as the
% "source" and TO the column as the "target." This is known as the
% input-row (IR) convention.
% Conversely, the design structure matrix (DSM) uses the input column (IC)
% convention and so the DSM is a transpose of cM. Also, the DSM does not
% calculate the accumulated number of interactions, it just indicates that
```

```
% a connection exists, as a one (1) and that if no interface exists,  
% indicated as a "0".
```

Builds the DSM.

```
DSM = MPdsm(cM.all);
```

Remove events that have no interactions

```
[DSM,removeEvents] = MPremoveEvents(DSM);
```

Prepare the the DSM for analysis, retaining the original file

```
DSM_matrix = DSM;  
DSM_matrix_original = DSM_matrix; % retain a version of the original DSM  
DSM_size = length(DSM);
```

Find the event labels for the DSM analysis, excluding the events removed

from consideration

```
DSMLABEL = MPEventsDSM(eventList, removeEvents, DSM_size);
```

Set-up the clustering parameters

```
Cluster_param = MPsetupDSM(DSM_size);
```

Run clustering

```
[Cluster_matrix, total_coord_cost, cost_history, old_data] = ...  
Cluster(DSM_matrix, Cluster_param);
```

Display graphs

```
MPgraphDSM;
```

```
Cluster # 1  
Processor_Function (19)  
interpret_as_not_object (22)  
send_merged_data (23)  
not_send_merged_data (29)  
interpret_as_unknown (31)  
interpret_as_object (39)  
  
Cluster # 2  
is_not_of_interest (10)  
Object_Function (11)  
present_observation (12)  
not_receive_effect (13)  
not_present_observation (35)  
is_of_interest (37)  
  
Cluster # 3  
Environment (1)  
Environment_Function (6)  
not_include_problem (7)  
success (8)  
include_problem (36)  
fail (41)  
  
Cluster # 4  
Actors (5)  
Actor (24)  
Actor_Function (25)  
not_receive_merged_data (28)  
  
Cluster # 5  
Sensor_Function (15)  
not_send_data (33)  
not_detect_object (34)
```

```
Cluster # 6
  Sensor_Function (15)
  detect_object (16)
  send_data (17)

Cluster # 7
  receive_effect (38)
  act (40)

Cluster # 8
  not_merge_data (30)
  not_receive_data (32)

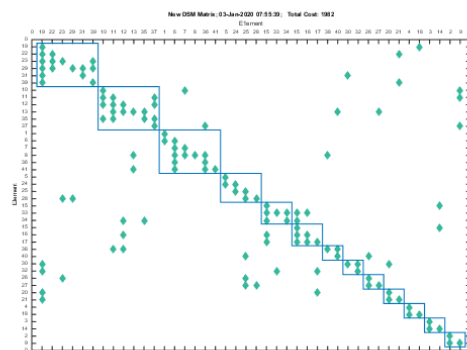
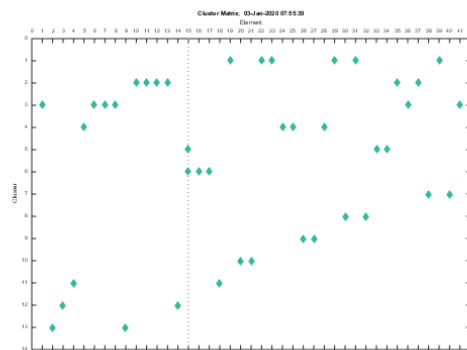
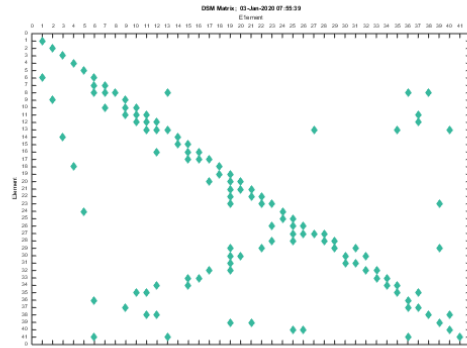
Cluster # 9
  receive_merged_data (26)
  not_act (27)

Cluster #10
  receive_data (20)
  merge_data (21)

Cluster #11
  Processors (4)
  Processor (18)

Cluster #12
  Sensors (3)
  Sensor (14)

Cluster #13
  Objects (2)
  Object (9)
```



Cluster Member List

03-Jan-2020 07:55:39

Cluster #1	Cluster #4
Processor_Function (19)	Actors (5)
interpret_as_not_object (22)	Actor (24)
send_merged_data (23)	Actor_Function (25)
not_send_merged_data (29)	not_receive_merged_data (28)
interpret_as_unknown (31)	
interpret_as_object (38)	Cluster #5
Cluster #2	* Sensor_Function (15)
is_not_of_interest (10)	not_send_data (33)
Object_Function (11)	not_delete_object (34)
present_observation (12)	Cluster #6
not_receive_effect (13)	* Sensor_Function (15)
not_present_observation (35)	detect_object (16)
is_of_interest (37)	send_data (17)
Cluster #3	Cluster #7
Environment (1)	receive_effect (38)
Environment_Function (6)	act (40)
not_include_problem (7)	
success (8)	
include_problem (36)	
fail (41)	

Cluster Member List

03-Jan-2020 07:55:39

Cluster #8	Cluster #13
not_merge_data (30)	Objects (2)
not_receive_data (32)	Object (9)
Cluster #9	
receive_merged_data (26)	
not_act (27)	
Cluster #10	
receive_data (20)	
merge_data (21)	
Cluster #11	
Processors (4)	
Processor (18)	
Cluster #12	
Sensors (3)	
Sensor (14)	

Print out the DSM matrices before and after clustering

function MPprintDSM(A, name, order, title, fout, fcsv)

```
MPprintDSM(place_diag(DSM_matrix_original, 1), cM, ...
DSMLABEL, DSM_labels, 'Initial', fout, fcsv);

MPprintDSM(place_diag(New_DSM_matrix, 1), cM, ...
DSMLABEL, New_DSM_labels, 'Updated', fout, fcsv);
```

Initial Design Structured Matrix (DSM):

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)
Environment(1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Objects(2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sensors(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Processors(4)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Actors(5)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Environment_Function(6)	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_include_problem(7)	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0
success(8)	0	0	0	0	0	9	8	0	0	0	0	0	8	0	0	0	0	0
Object(9)	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
is_not_of_interest(10)	0	0	0	0	0	0	8	0	8	0	0	0	0	0	0	0	0	0
Object_Function(11)	0	0	0	0	0	0	0	0	18	8	0	0	0	0	0	0	0	0
present_observation(12)	0	0	0	0	0	0	0	0	0	7	16	0	0	0	0	0	0	0
not_receive_effect(13)	0	0	0	0	0	0	0	0	0	0	17	15	0	0	0	0	0	0
Sensor(14)	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sensor_Function(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0
detect_object(16)	0	0	0	0	0	0	0	0	0	0	0	14	0	0	14	0	0	0
send_data(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	12	12	0	0
Processor(18)	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Processor_Function(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	18
receive_data(20)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
merge_data(21)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
interpret_as_not_object(22)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
send_merged_data(23)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Actor(24)	0	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0	0
Actor_Function(25)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
receive_merged_data(26)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_act(27)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_receive_merged_data(28)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_send_merged_data(29)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_merge_data(30)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
interpret_as_unknown(31)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_receive_data(32)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0
not_send_data(33)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	2	0	0
not_detect_object(34)	0	0	0	0	0	0	0	0	0	0	0	2	0	0	4	0	0	0
not_present_observation(35)	0	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	0
include_problem(36)	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	0	0	0
is_of_interest(37)	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0
receive_effect(38)	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
interpret_as_object(39)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
act(40)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
fail(41)	0	0	0	0	0	9	0	0	0	0	0	0	9	0	0	0	0	0

Updated Design Structured Matrix (DSM):

	(19)	(22)	(23)	(29)	(31)	(39)	(10)	(11)	(12)	(13)	(35)	(37)	(1)	(6)	(7)	(8)	(36)	(41)
Processor_Function(19)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
interpret_as_not_object(22)	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
send_merged_data(23)	6	2	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0
not_send_merged_data(29)	12	1	0	0	10	1	0	0	0	0	0	0	0	0	0	0	0	0
interpret_as_unknown(31)	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
interpret_as_object(39)	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
is_not_of_interest(10)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0
Object_Function(11)	0	0	0	0	0	0	8	0	0	0	0	10	0	0	0	0	0	0
present_observation(12)	0	0	0	0	0	0	7	16	0	0	0	9	0	0	0	0	0	0
not_receive_effect(13)	0	0	0	0	0	0	0	17	15	0	2	0	0	0	0	0	0	0
not_present_observation(35)	0	0	0	0	0	0	1	2	0	0	0	1	0	0	0	0	0	0
is_of_interest(37)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0
Environment(1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Environment_Function(6)	0	0	0	0	0	0	0	0	0	0	0	0	18	0	0	0	0	0
not_include_problem(7)	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	0	0	0
success(8)	0	0	0	0	0	0	0	0	0	8	0	0	0	9	8	0	1	0
include_problem(36)	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	0	0	0
fail(41)	0	0	0	0	0	0	0	0	9	0	0	0	0	9	0	0	9	0
Actors(5)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Actor(24)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Actor_Function(25)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_receive_merged_data(28)	0	0	2	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sensor_Function(15)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_send_data(33)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_detect_object(34)	0	0	0	0	0	0	0	0	2	0	2	0	0	0	0	0	0	0
Sensor(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
detect_object(16)	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0
send_data(17)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
receive_effect(38)	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
act(40)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_merge_data(30)	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

not_receive_data(32)	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
receive_merged_data(26)	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
not_act(27)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
receive_data(20)	10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
merge_data(21)	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Processors(4)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Processor(18)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sensors(3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Sensor(14)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Objects(2)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Object(9)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Published with MATLAB® R2014b

List of References

- [1] B. P. Zeigler, “A note on promoting positive emergence and managing negative emergence in systems of systems,” *The Journal of Defense Modeling and Simulation*, vol. 13, no. 1, pp. 133–136, 2016. [Online]. Available: <https://doi.org/10.1177/1548512915620580>
- [2] L. B. Rainey and M. Jamshidi, *Engineering Emergence: A Modeling and Simulation Approach*. Boca Raton, FL: CRC Press, 2018.
- [3] F. P. Brooks, *The Mythical Man-Month*. Boston, MA: Addison-Wesley, 2010.
- [4] J. Quartuccio, K. Giammarco, and M. Auguston, “Identifying decision patterns using Monterey Phoenix,” in *12th International Conference on System of Systems Engineering (SoSE)*, 2017. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/7994952>
- [5] J. Quartuccio, K. Giammarco, and M. Auguston, “Deriving probabilities from behavior models defined in Monterey Phoenix,” in *12th International Conference on System of Systems Engineering (SoSE)*, 2017.
- [6] J. Quartuccio and K. Giammarco, “A model-based approach to investigate emergent behaviors in systems of systems,” in *Engineering Emergence: A Modeling and Simulation Approach*, L. Rainey and M. Jamshidi, Eds. CRC Press, 2018.
- [7] *Pre-Milestone a and Early-Phase Systems Engineering : A Retrospective Review and Benefits for Future Air Force Acquisition*, Air Force Studies Board, National Research Council, Washington, DC, 2008. [Online]. Available: <http://site.ebrary.com/lib/nps/docDetail.action?docID=10495470>
- [8] *CORE Architecture Definition Guide*, 2015, Accessed October 8, 2018. [Online]. Available: <http://www.vitechcorp.com/support/documentation/core/900/ArchitectureDefinitionGuide.pdf>
- [9] Monterey Phoenix Home Page. (2018). Naval Postgraduate School. [Online]. Available: <https://wiki.nps.edu/display/MP/Monterey+Phoenix+Home>
- [10] M. Auguston, “Software architecture built from behavior models,” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 5, pp. 1–15, 2009. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/1598732.1598733>
- [11] M. Auguston, “Monterey Phoenix, or how to make software architecture executable,” in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pp. 1031–1040. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/1639950.1640075>

- [12] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, 2012.
- [13] D. Jackson, “Alloy: A lightweight object modelling notation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, Apr. 2002. [Online]. Available: <http://doi.acm.org.libproxy.nps.edu/10.1145/505145.505149>
- [14] G. E. Moore, “The future of integrated electronics,” *Electronics*, 1965.
- [15] K. Giammarco and M. Auguston, “Monterey phoenix main principles and advantages,” 2015, Accessed July 16, 2015. [Online]. Available: <https://wiki.nps.edu/display/MP/Main+Principles+and+Advantages>
- [16] D. Jackson, “Automating first-order relational logic,” *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 6, pp. 130–139, Nov 2000, SIGSOFT ’00/FSE-8 Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications. [Online]. Available: <http://doi.acm.org.libproxy.nps.edu/10.1145/357474.355063>
- [17] M. Syed, *Black Box Thinking: the surprising truth about success*. John Murray, 2015.
- [18] J. Dahmann, A. Markina-Khusid, A. Doren, T. Wheeler, M. Cotter, and M. Kelley, “SysML executable systems of system architecture definition: A working example,” in *2017 Annual IEEE International Systems Conference (SysCon)*. IEEE, 2017, pp. 1–6.
- [19] L. Antul, S. Ricks, L. M. K. Cho, M. Cotter, R. B. Jacobs, A. Markina-Khusid, J. Kamenetsky, J. Dahmann, and H. T. Tran, “Toward scaling model-based engineering for systems of systems,” in *2018 IEEE Aerospace Conference*. IEEE, 2018, pp. 1–9.
- [20] M. W. Maier, “The role of modeling and simulation in system of systems development,” *Modeling and simulation support for system of systems engineering applications*, pp. 11–41, 2015. [Online]. Available: <https://onlinelibrary.wiley.com/doi/10.1002/9781118501757.ch2>
- [21] S. Friedenthal, R. Griego, and M. Sampson, “INCOSE model based systems engineering (MBSE) initiative,” in *INCOSE 2007 Symposium*, 2007. [Online]. Available: https://www.researchgate.net/profile/Mark_Sampson3/publication/267687693_INCOSE_Model_Based_Systems_Engineering_MBSE_Initiative/links/54ca7c290cf22f98631b167e.pdf
- [22] L. Von Bertalanffy, “An outline of general system theory.” *British Journal for the Philosophy of science*, 1950. [Online]. Available: <https://psycnet.apa.org/record/1951-02734-001>
- [23] L. V. Bertalanffy, “The history and status of general systems theory,” *The Academy of Management Journal*, vol. 15, no. 4, pp. 407–426, 1972. [Online]. Available: <http://www.jstor.org/stable/255139>
- [24] L. R. Troncale, “Linkage propositions between fifty principal systems concepts,” in *Applied General Systems Research*. Boston, MA: Springer, 1978, pp. 29–52. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4757-0555-3_2

- [25] L. Troncale, "Can a theory that integrates the natural systems sciences help systems engineering of defense against security threats?" in *2011 Eighth International Conference on Information Technology: New Generations*. IEEE, 2011, pp. 947–952. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/5945362>
- [26] R. Cloutier, "Applicability of patterns to architecting complex systems," Ph.D. dissertation, Stevens Institute of Technology, Hoboken, NJ, 2006, Dept of Sys. Eng. and Eng. Mgt.
- [27] R. J. Cloutier, J. Boardman, and D. Verma, "Application of patterns to systems engineering and architecting," in *Proceedings of the 16th Annual International Symposium of INCOSE*, 2006. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2006.tb02791.x>
- [28] D. Rousseau, J. Wilby, J. Billingham, and S. Blachfellner, "The scope and range of general systems transdisciplinarity," *Systema: connecting matter, life, culture and technology*, vol. 4, no. 1, pp. 48–60, 2016.
- [29] D. Rousseau, "Three general systems principles and their derivation: Insights from the philosophy of science applied to systems concepts," in *Disciplinary Convergence in Systems Engineering Research*. Springer, 2018, pp. 665–681. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-62217-0_46
- [30] D. Hammond. (2010). *The Science of Synthesis: Exploring the Social Implications of General Systems Theory*. Boulder, CO. [Online]. Available: <https://play.google.com/books/reader?id=TBvjCwAAQBAJ&lr=&printsec=frontcover&pg=GBS.PP1>
- [31] L. Von Bertalanffy, *Problems of life; an evaluation of modern biological thought*. New York, NY: Wiley, 1952.
- [32] L. Friendshuh and L. Troncale, "Identifying fundamental systems processes for a general theory of systems," in *Proceedings of the 56th Annual Conference, International Society for the Systems Sciences (ISSS)*, San Jose, CA, 2012, pp. 15–20.
- [33] L. Von Bertalanffy, "General system theory," *New York*, vol. 41973, no. 1968, p. 40, 1968.
- [34] K. Giammarco and L. Troncale, "Modeling isomorphic systems processes using Monterey Phoenix," *Systems*, vol. 6, no. 2, p. 18, 2018. [Online]. Available: <https://www.mdpi.com/2079-8954/6/2/18>
- [35] J. Boyd, "Patterns of conflict," December 1986, unpublished study.
- [36] J. R. Boyd, "Organic design for command and control," *A discourse on winning and losing*, 1987. [Online]. Available: https://www.airspacepower.net/JRB/organic_design.pdf
- [37] R. J. Cloutier and D. Verma, "Applying the concept of patterns to systems architecture," *Systems Engineering*, vol. 10, no. 2, pp. 138–154, 2007. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.20066>

- [38] A. Kossiakoff and W. Sweet, *Systems Engineering Principles and Practice*. Hoboken, NJ: Wiley, 2003.
- [39] G. D. Wells and A. P. Sage, "Engineering of a system of systems," in *System of Systems Engineering: Innovations for the 21st Century*, M. Jamshidi, Ed. Hoboken, NJ: John Wiley & Sons, 2009, ch. Chapter 3, pp. 44–76.
- [40] "Introduction to systems engineering," *Defense Acquisition Guidebook(DAG)*, 2000.
- [41] C. Calvano and P. John, "Systems engineering in an age of complexity," *IEEE Engineering Management Review*, vol. 32, no. 4, pp. 29–37, 2004. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.10054>
- [42] M. M. Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York, NY: Simon and Schuster, 1993.
- [43] S. H. Strogatz, *Sync: How Order Emerges from Chaos in the Universe, Nature, and Daily Life*. New York, NY: Hyperion, 2003.
- [44] J. H. Miller and S. E. Page, *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton, NJ: Princeton University Press, 2009.
- [45] M. Mitchell, *Complexity: A Guided Tour*. New York, NY: Oxford University Press, 2009.
- [46] D. Jackson, G. Sedrick, and K. Tayeb, "Algorithmic development of effectiveness prediction for system of systems," in *41st Southeastern Symposium on System Theory*, March 2009, pp. 164–168. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4806840>
- [47] C. B. Keating, "Emergence in system of systems," in *System of Systems Engineering: Innovations for the 21st Century*, M. Jamshidi, Ed. Hoboken, NJ: Wiley, 2009, ch. Chapter 7, pp. 169–190.
- [48] D. Rhodes, A. Ross, and D. Nightingale, "Architecting the system of systems enterprise: Enabling constructs and methods from the field of engineering systems," in *3rd Annual IEEE Systems Conference*, March 2009, pp. 190–195. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4815796>
- [49] M. DiMario, J. Boardman, and B. Sauser, "System of systems collaborative formation," *Systems Journal, IEEE*, vol. 3, no. 3, pp. 360–368, Sept 2009. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/5229218>
- [50] R. Volkert, C. Jackson, J. Stracener, and J. Yu, "Development and extension of a deterministic system of systems performance prediction methodology for an acknowledged system of systems," Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS-AM-12-C9P02R02-044, 2012. [Online]. Available: <https://apps.dtic.mil/docs/citations/ADA563330>
- [51] J. H. Holland, *Hidden order: How Adaptation Builds Complexity*. New York, NY: Basic Books, 1995, no. 003.7 H6.

- [52] J. H. Holland, *Emergence: From Chaos to Order*. New York, NY: Basic Books, 2000.
- [53] O. T. Holland, "Partitioning method for emergent behavior systems modeled by agent-based simulations," Ph.D. dissertation, Dept. of Modeling, Simul., & Vis. Eng., Old Dominion University, 2012. [Online]. Available: https://digitalcommons.odu.edu/msve_etds/36/
- [54] R. Gore, P. F. Reynolds Jr, L. Tang, and D. C. Brogan, "Explanation exploration: Exploring emergent behavior," in *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 2007, pp. 113–122. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4262797>
- [55] R. Gore and P. F. Reynolds Jr, "Applying causal inference to understand emergent behavior," in *Proceedings of the 40th Conference on Winter Simulation*. Winter Simulation Conference, 2008, pp. 712–721. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4736133>
- [56] O. T. Holland, "Towards a lexicon for the modeling and simulation of emergent behavior systems," in *Proceedings of 2008 Spring SIW/BRIMS Conference*, no. 08S-SIW-058, Providence, RI, April 2008.
- [57] O. T. Holland, "Taxonomy for the modeling and simulation of emergent behavior systems," in *Proceedings of the 2007 Spring Simulation Multiconference*. San Diego, CA: Society for Computer Simulation International, 2007, vol. 2, pp. 28–35. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1404680.1404684>
- [58] C. W. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM Siggraph Computer Graphics*, vol. 21, no. 4, pp. 25–34, 1987. [Online]. Available: <https://dl.acm.org/doi/abs/10.1145/37401.37406>
- [59] A. Pentland and A. Liu, "Modeling and prediction of human behavior," *Neural computation*, vol. 11, no. 1, pp. 229–242, 1999. [Online]. Available: <https://www.mitpressjournals.org/doi/abs/10.1162/089976699300016890>
- [60] J. Kantert, S. Tomforde, M. Kauder, R. Scharrer, S. Edenhofer, J. Hähner, and C. Müller-Schloer, "Controlling negative emergent behavior by graph analysis at runtime," *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 2, June 2016. [Online]. Available: <https://doi.org/10.1145/2890507>
- [61] C. L. Bouwens, "Systems geometry: A methodology for analyzing emergent system of systems behaviors," Ph.D. dissertation, Dept. of Ind. Eng. and Mgt. Sys., University of Central Florida, Orlando, FL, 2013.
- [62] G. E. P. Box, "Science and statistics," *Journal of the American Statistical Association*, vol. 71, no. 356, pp. 791–799, 1976. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/01621459.1976.10480949>

- [63] J. Zachman *et al.*, “A framework for information systems architecture,” *IBM systems journal*, vol. 26, no. 3, pp. 276–292, 1987. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/5387671>
- [64] D. M. Buede, *The engineering design of systems: models and methods*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2009.
- [65] A. Borshchev, *The Big Book of Simulation Modeling: Multimethod Modeling with AnyLogic* 6. AnyLogic North America, 2013.
- [66] M. Auguston, *Monterey Phoenix, System and Software Architecture and Workflow Modeling Language Manual*, 4th ed., Naval Postgraduate School, Monterey CA, May 2019. [Online]. Available: <https://wiki.nps.edu/display/MP/Documentation>
- [67] B. P. Zeigler, *Theory of Modeling and Simulation*. Hoboken, NJ: John Wiley & Sons, 1976.
- [68] O. A. Yakimenko, *Engineering computations and modeling in MATLAB/Simulink*. Reston, VA: American Institute of Aeronautics and Astronautics, 2011.
- [69] B. S. Blanchard, W. J. Fabrycky, and W. J. Fabrycky, *Systems engineering and analysis*. Englewood Cliffs, NJ: Prentice Hall, 1990, vol. 4.
- [70] L. Delligatti, *SysML distilled: A brief guide to the systems modeling language*. Upper Saddle River, NJ: Addison-Wesley, 2013.
- [71] M. W. Maier and E. Rechtin, *The art of systems architecting*. Boca Raton, FL: CRC Press, 2009.
- [72] J. A. Zachman. (2008). A Concise Definition of the Zachman Framework. [Online]. Available: <https://www.zachman.com/about-the-zachman-framework>
- [73] *Department of Defense Architecture Framework (DoDAF) Version 2.02*, Department of Defense, Washington, DC, 2010. [Online]. Available: <https://dodcio.defense.gov/Library/DoD-Architecture-Framework/>
- [74] *The MoD Architecture Framework*, Ministry of Defence, Whitehall, London, 2012. [Online]. Available: <https://www.gov.uk/guidance/mod-architecture-framework>
- [75] H. W. Ang, D. Nicholson, and B. Mercer, “Improving the practice of dod architecting with the architecture specification model,” DTIC Document, Tech. Rep., 2005. [Online]. Available: <https://apps.dtic.mil/docs/citations/ADA456395>
- [76] B. Mercer, “Enabling executable architecture by improving the foundations of dod architecting,” in *Symposium on International Collaborative Technologies and Systems*, May 2008, pp. 558–559. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4543979>
- [77] Object Modeling Group. [Online]. Available: <https://www.omg.org>. Accessed September 22, 2018.

- [78] *OMG Unified Modeling Language (UML)*, Object Modeling Group Std. formal/2015-03-01, Rev. 2.5, March 2015. [Online]. Available: <http://www.omg.org/spec/UML/2.5/PDF/>
- [79] *OMG Systems Modeling Language (SysML)*, Object Modeling Group Std., Rev. 1.5, May 2017. [Online]. Available: <http://www.omg.org/spec/SysML/1.5/>
- [80] M. Hause, “Rebuilding the tower of babel the case for a unified architecture framework,” in *INCOSE International Symposium*, no. 1. Wiley Online Library, 2013, vol. 23, pp. 1460–1474. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2013.tb03099.x>
- [81] M. Hause, “Unified architecture framework (UAF) for system of systems modeling,” *System of Systems Engineering Collaborators Information Exchange (SoSECIE)*, 2016. [Online]. Available: <https://www.acq.osd.mil/se/webinars/2016-04-12-SoSECIE-Hause-brief.pdf>
- [82] M. Hause, “Using MBSE to evaluate and protect the electrical grid as a system of systems,” in *INCOSE International Symposium*, no. 1. Wiley Online Library, 2017, vol. 27, pp. 597–612. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2017.00381.x>
- [83] *UML Profile-Based Integrated Architecture, Reference Guide*, 8th ed., IBM, June 2010, Accessed October 13, 2016. [Online]. Available: <http://www-01.ibm.com/support/docview.wss?uid=swg27019099&aid=1>
- [84] About the Unified Profile for DoDAF and MoDAF Specification Version 2.1. (2013, July). Object Modeling Group. [Online]. Available: <https://www.omg.org/spec/UPDM/About-UPDM/>
- [85] *OMG Unified Architecture Framework (UAF)*, Object Modeling Group Std., October 2017. [Online]. Available: <https://www.omg.org/spec/UAF/About-UAF/>
- [86] “World in motion: Systems engineering vision 2025,” *International Council on Systems Engineering*, 2014.
- [87] R. Arp, B. Smith, and A. D. Spear, *Building ontologies with basic formal ontology*. Cambridge, MA: MIT Press, 2015.
- [88] B. Smith, “Basic concepts of formal ontology,” 1998. [Online]. Available: <https://philpapers.org/rec/SMITBT>
- [89] B. Smith, “On classifying material entities in basic formal ontology,” 2012. [Online]. Available: <https://philpapers.org/rec/SMIOCM>
- [90] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein, “Oil in a nutshell,” in *International Conference on Knowledge Engineering and Knowledge Management*. Springer, 2000, pp. 1–16. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-39967-4_1

- [91] D. Fensel, F. Van Harmelen, I. Horrocks, D. L. McGuinness, and P. F. Patel-Schneider, "Oil: An ontology infrastructure for the semantic web," *IEEE intelligent systems*, vol. 16, no. 2, pp. 38–45, 2001. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/920598>
- [92] S. Decker, S. Melnik, F. Van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks, "The semantic web: The roles of xml and rdf," *IEEE Internet computing*, vol. 4, no. 5, pp. 63–73, 2000. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/877487>
- [93] J. Davies, D. Fensel, and F. Van Harmelen, *Towards the semantic web: ontology-driven knowledge management*. Hoboken, NJ: John Wiley & Sons, 2003.
- [94] The DARPA Agent Markup Language. [Online]. Available: <http://www.daml.org/>. Accessed September 22, 2018.
- [95] *Web Ontology Language (OWL)*, W3C, December 2012. [Online]. Available: <https://www.w3.org/TR/owl2-overview/>
- [96] D. L. McGuinness, F. Van Harmelen *et al.*, "OWL web ontology language overview," *W3C*, vol. 10, no. 10, p. 2004, 2004.
- [97] S. Bechhofer, F. Van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, L. A. Stein *et al.*, "OWL web ontology language reference," *W3C*, vol. 10, no. 02, 2004.
- [98] G. Antoniou and F. Van Harmelen, "Web ontology language: OWL," in *Handbook on ontologies*. Springer, 2004, pp. 67–92. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-24750-0_4
- [99] A. Gangemi, "Ontology design patterns for semantic web content," in *International semantic web conference*. Springer, 2005, pp. 262–276. [Online]. Available: https://link.springer.com/chapter/10.1007/11574620_21
- [100] A. Gangemi and V. Presutti, "Ontology design patterns," in *Handbook on ontologies*. Springer, 2009, pp. 221–243. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-92673-3_10
- [101] V. Presutti, E. Blomqvist, E. Daga, and A. Gangemi, "Pattern-based ontology design," in *Ontology Engineering in a Networked World*. Springer, 2012, pp. 35–64. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-24794-1_3
- [102] A. Maedche and S. Staab, "Ontology learning for the semantic web," *IEEE Intelligent systems*, vol. 16, no. 2, pp. 72–79, 2001. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/920602>

- [103] A. Pease, I. Niles, and J. Li, “The suggested upper merged ontology: A large ontology for the semantic web and its applications,” in *Working notes of the AAAI-2002 workshop on ontologies and the semantic web*, 2002, vol. 28, pp. 7–10. [Online]. Available: <https://www.aaai.org/Papers/Workshops/2002/WS-02-11/WS02-11-011.pdf>
- [104] C. Alexander, *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press, 1977.
- [105] C. Alexander, *The Timeless Way of Building*. New York, NY: Oxford University Press, 1979, vol. 1.
- [106] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA: Addison Wesley, 1995.
- [107] W. Schindel, T. Peterson, G. Shuebrook, L. VanZandt, and R. Welling. (2015). MBSE methodology summary: Pattern-Based Systems Engineering (PBSE), based on S* MBSE models, version 1.5.5A. [Online]. Available: <http://www.omgwiki.org/MBSE/doku.php?id=mbse:pbse>
- [108] W. D. Schindel, “What is the smallest model of a system?” in *INCOSE International Symposium*, no. 1. Wiley Online Library, 2011, vol. 21, pp. 99–113. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2011.tb01188.x>
- [109] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. Boston, MA: Academic Press, 2000.
- [110] B. Zeigler, H. S. Sarjoughian, R. Duboz, and J.-C. Soulie, *Guide to Modeling and Simulation of Systems of Systems*, L. G. Birta, Ed. London, England: Springer Science & Business Media, 2012.
- [111] C. B. Jones, *Software Development. A Rigorous Approach*. Englewood Cliffs, NJ: Prentice Hall, 1980.
- [112] C. B. Jones, *Development Methods for Computer Programs Including a Notion of Interference*. Oxford, England: Oxford University Computing Laboratory, 1981.
- [113] C. B. Jones, “A rigorous approach to formal methods,” *ACM Comput. Surv.*, vol. 28, no. 4es, p. 121, 1996. [Online]. Available: https://www.researchgate.net/profile/Cliff_Jones/publication/243611610_A_rigorous_approach_to_formal_methods/links/54085cf40cf2bba34c275bdc.pdf
- [114] D. Jackson, “Lightweight formal methods,” in *International Symposium of Formal Methods Europe*. Springer, 2001, pp. 1–1. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45251-6_1
- [115] K. Giammarco, “A formal method for assessing architecture model and design maturity using domain-independent patterns,” *Procedia Computer Science*, vol. 28, pp. 555–564, 2014. [Online]. Available: <https://calhoun.nps.edu/handle/10945/63778>

- [116] P. Zimmerman, T. Gilbert, and F. Salvatore, “Digital engineering transformation across the department of defense,” *The Journal of Defense Modeling and Simulation*, pp. 325–338, 2017. [Online]. Available: <https://journals.sagepub.com/doi/abs/10.1177/1548512917747050>
- [117] Joint Chiefs of Staff, “Department of defense dictionary of military and associated terms,” Department of Defense, Tech. Rep. Joint Publication 1-02, September 2018, Accessed October 1, 2018. [Online]. Available: <http://www.jcs.mil/Doctrine/DOD-Terminology/>
- [118] M. A. Bone, M. R. Blackburn, D. H. Rhodes, D. N. Cohen, and J. A. Guerrero, “Transforming systems engineering through digital engineering,” *The Journal of Defense Modeling and Simulation*, pp. 339–355, 2018. [Online]. Available: <https://journals.sagepub.com/doi/full/10.1177/1548512917751873>
- [119] M. Blackburn, D. Verma, R. Dillon-Merrill, R. Blake, M. Bone, B. Chell, R. Dove, J. Dzielski, P. Grogan, S. Hoffenson *et al.*, “Transforming systems engineering through model centric engineering,” Stevens Institute of Technology, Hoboken, NJ, Tech. Rep., 2018. [Online]. Available: <https://apps.dtic.mil/dtic/tr/fulltext/u2/1058354.pdf>
- [120] B. Kruse, M. A. Bone, and M. Blackburn, “Systems engineering research center (SERC),” 2018. [Online]. Available: https://ndiastorage.blob.core.usgovcloudapi.net/ndia/2018/systems/Wed_21466_Kruse.pdf
- [121] DAU Credentialing Pilot Program, Digital Engineering: DOD Consumer Credential. (2019). [Online]. Available: <https://asc.army.mil/web/news-dau-credentialing-pilot-program-launches-fall-2019/>
- [122] K. Giammarco, “Architecture modeling software analytics - model quality and maturity using automated tools,” in *12th International Conference on System of Systems Engineering (SoSE)*, 2017. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/7994962>
- [123] *Systems and Software Engineering - System Life Cycle Processes*. New York, NY: ISO/IEC/IEEE 15288, 2015.
- [124] *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*, v 1.4, Hoboken, NJ, 2015, Accessed July 11, 2015. [Online]. Available: [https://www.sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_\(SEBoK\)](https://www.sebokwiki.org/wiki/Guide_to_the_Systems_Engineering_Body_of_Knowledge_(SEBoK))
- [125] M. Jamshidi, “System of systems engineering - an overview,” in *Engineering Emergence: A Modeling and Simulation Approach*, L. Rainey and M. Jamshidi, Eds. Boca Raton, FL: CRC Press, 2018.
- [126] M. W. Maier, “Architecting principles for systems-of-systems,” *INCOSE International Symposium*, vol. 6, no. 1, pp. 565–573, 1996. [Online]. Available: <http://dx.doi.org/10.1002/j.2334-5837.1996.tb02054.x>

- [127] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998. [Online]. Available: [http://dx.doi.org/10.1002/\(SICI\)1520-6858\(1998\)1:4<267::AID-SYS3>3.0.CO;2-D](http://dx.doi.org/10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D)
- [128] J. Dahmann and K. Baldwin, "Understanding the current state of us defense systems of systems and the implications for systems engineering," in *Systems Conference, 2008 2nd Annual IEEE*, April 2008, pp. 1–7. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4518994>
- [129] J. S. Dahmann, "Systems engineering for department of defense systems of systems," in *System of Systems Engineering: Innovations for the 21st Century*, M. Jamshidi, Ed. Hoboken, NJ: John Wiley & Sons, 2009, pp. 218–231.
- [130] J. H. Holland, "Complex adaptive systems," *Daedalus*, pp. 17–30, 1992. [Online]. Available: <https://www.jstor.org/stable/20025416?seq=1>
- [131] J. H. Holland, "Studying complex adaptive systems," *Journal of Systems Science and Complexity*, vol. 19, no. 1, pp. 1–8, 2006. [Online]. Available: <https://link.springer.com/article/10.1007/s11424-006-0001-z>
- [132] C. W. Johnson, "What are emergent properties and how do they affect the engineering of complex systems?" *Reliability Engineering & System Safety*, vol. 91, no. 12, pp. 1475 – 1481, 2006. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0951832006000330>
- [133] "Abstract," *Merriam-Webster*. Accessed June 10, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/abstract>
- [134] "Model," *Merriam-Webster*. Accessed June 9, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/model>
- [135] "Simulate," *Merriam-Webster*. Accessed June 9, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/simulate>
- [136] "Simulation," *Merriam-Webster*. Accessed June 9, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/simulation>
- [137] "What is a model?" March 2013, Accessed July 5, 2017. [Online]. Available: <http://model-based-systems-engineering.com/2013/03/15/what-is-a-model/>
- [138] "Syntax," *Merriam-Webster*. Accessed August 3, 2019. [Online]. Available: <https://www.merriam-webster.com/dictionary/syntax>
- [139] "Semantic," *Merriam-Webster*. Accessed October 8, 2018. [Online]. Available: <https://www.merriam-webster.com/dictionary/semantic>
- [140] "Ontology," *Merriam-Webster*. Accessed October 8, 2018. [Online]. Available: <https://www.merriam-webster.com/dictionary/ontology>

- [141] “Joint simulation system composability task force final report,” *JSIMS*, Fort Monroe, VA. 1997. [Online]. Available: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a358596.pdf>
- [142] S. M. Harkrider and W. Lunceford, “Modeling and simulation composability,” in *I/ITSEC: Proceedings of the Interservice/Industry Training, Simulation and Education Conference*, 1999.
- [143] D. R. Pratt, L. Ragusa, and S. von der Lippe, “Composability as an architecture driver,” in *Proceedings of the 1999 I/ITSEC Conference*, 1999.
- [144] S. Kasputis and H. C. Ng, “Model composability: Formulating a research thrust: Composable simulations,” in *Proceedings of the 32Nd Conference on Winter Simulation (WSC ’00)*. San Diego, CA: Society for Computer Simulation International, 2000, pp. 1577–1584. [Online]. Available: <http://dl.acm.org/citation.cfm?id=510378.510609>
- [145] M. D. Petty and E. W. Weisel, “A composability lexicon,” in *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, 2003, vol. 2003, pp. 181–187.
- [146] P. K. Davis and R. H. Anderson, “Improving the composability of department of defense models and simulations,” Rand Corp., Santa Monica, CA, Tech. Rep., 2003. [Online]. Available: <https://apps.dtic.mil/docs/citations/ADA451702>
- [147] “Behavior,” *Merriam-Webster*. Accessed June 9, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/behavior>
- [148] “Activity,” *Merriam-Webster*. Accessed June 10, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/activity>
- [149] “Action,” *Merriam-Webster*. Accessed June 10, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/action>
- [150] “Event,” *Merriam-Webster*. Accessed June 10, 2017. [Online]. Available: <https://www.merriam-webster.com/dictionary/event>
- [151] H. Pospesel, *Introduction to Logic: Propositional Logic*. Upper Saddle River, NJ: Prentice Hall, 1999.
- [152] H. Pospesel, *Introduction to Logic: Predicate Logic*. Upper Saddle River, NJ: Prentice Hall, 2003.
- [153] “Process,” *Merriam-Webster*. Accessed July 7, 2019. [Online]. Available: <https://www.merriam-webster.com/dictionary/process>
- [154] “Process,” *Merriam-Webster*. Accessed July 20, 2019. [Online]. Available: <https://www.merriam-webster.com/dictionary/isomorphic>
- [155] “Pattern,” *Merriam-Webster*. Accessed November 24, 2018. [Online]. Available: <https://www.merriam-webster.com/dictionary/pattern>

- [156] M. Fowler, *Analysis Patterns: Reusable Object Models*. Menlo Park, CA: Addison-Wesley Professional, 1997.
- [157] *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, Version 4*. International Committee on Systems Engineering, Wiley, 2015. [Online]. Available: <https://books.google.com/books?id=ViDyCQAAQBAJ>
- [158] X. Zhang, E. Miller-Hooks, and K. Denny, “Assessing the role of network topology in transportation network resilience,” *Journal of Transport Geography*, vol. 46, pp. 35–45, 2015. [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/S0966692315000794>
- [159] *Defense Acquisition Guidebook*, Defense Acquisition University, Fort Belvoir, VA, June 2013, p. 1248. [Online]. Available: <https://acc.dau.mil/CommunityBrowser.aspx?id=511646&lang=en-US>
- [160] T. R. Browning, “Applying the design structure matrix to system decomposition and integration problems: a review and new directions,” *IEEE Transactions on Engineering management*, vol. 48, no. 3, pp. 292–306, 2001. [Online]. Available: <https://ieeexplore.ieee.org/document/946528>
- [161] S. D. Eppinger and T. R. Browning, *Design Structure Matrix Methods and Applications*. Cambridge, MA: MIT Press, 2012.
- [162] R. E. Thebeau, “Knowledge management of system interfaces and interactions from product development processes,” Master’s thesis, Dept. of Eng. and Mgt., MIT, Cambridge, MA, 2001. [Online]. Available: <https://dspace.mit.edu/handle/1721.1/29168>
- [163] L. Troncale, “Systems processes and pathologies: Creating an integrated framework for systems science,” in *INCOSE International Symposium*, no. 1. Wiley Online Library, 2013, vol. 23, pp. 1330–1353. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2013.tb03091.x>
- [164] M. Auguston, “Behavior models for software architecture,” Naval Postgraduate School, Monterey, CA, Tech. Rep. NPS-CS-14-003, 2014. [Online]. Available: <https://calhoun.nps.edu/handle/10945/43851>
- [165] K. Giammarco and M. Auguston, “Well, you didnt say not to! a formal systems engineering approach to teaching an unruly architecture good behavior,” *Procedia Computer Science*, vol. 20, pp. 277 – 282, 2013. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050913010739>
- [166] B. W. Boehm, “A spiral model of software development and enhancement,” *Computer*, no. 5, pp. 61–72, 1988. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/59>

- [167] S. D. Eppinger, D. E. Whitney, R. P. Smith, and D. A. Gebala, “A model-based method for organizing tasks in product development,” *Research in Engineering Design*, vol. 6, no. 1, pp. 1–13, 1994. [Online]. Available: <https://link.springer.com/article/10.1007/BF01588087>
- [168] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*. New York, NY: Wiley Interscience, 2001.
- [169] R. Coram, *Boyd: The Fighter Pilot Who Changed the Art of War*. New York, NY: Little, Brown and Co., 2002.
- [170] “Qualities of great models,” Short Course: Model-Based Systems Engineering: Documentation and Analysis, Massachusetts Institute of Technology, 2017, Accessed June 11, 2017. [Online]. Available: <https://mitprofessionalx.mit.edu/courses/course-v1:MITProfessionalXSysEngx31T2017/>
- [171] F. V. Jensen, *An Introduction to Bayesian networks*. London, England: UCL Press, 1996, vol. 210.
- [172] F. V. Jensen, *Bayesian Networks and Decision Graphs. Series for Statistics for Engineering and Information Science*. Berlin, Germany: Springer Science & Business Media, 2007. [Online]. Available: <https://books.google.com/books>
- [173] F. V. Jensen and T. D. Nielsen, *Causal and Bayesian Networks*. New York, NY: Springer New York, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-68282-2_2
- [174] A. Kolmogorov, *Foundations of the Theory of Probability*. Mansfield Centre, CT: Martino Publishing, 1956, English translation by N. Morrison.
- [175] J. L. Doob, “The development of rigor in mathematical probability (1900–1950),” *The American Mathematical Monthly*, vol. 103, no. 7, pp. 586–595, 1996. [Online]. Available: <https://www.tandfonline.com/doi/abs/10.1080/00029890.1996.12004791?journalCode=uamm20>
- [176] E. W. Weisstein, “Probability axioms,” 2017. [Online]. Available: <http://mathworld.wolfram.com/ProbabilityAxioms.html>
- [177] M. Auguston, *Monterey Phoenix, System and Software Architecture and Workflow Modeling Language Manual (version 3.5)*, 3rd ed., Naval Postgraduate School, Monterey California, July 2018. [Online]. Available: <https://wiki.nps.edu/display/MP/Documentation>
- [178] K. Fong, “What we can learn from fatal mistakes in surgery,” *British Broadcasting Corporation*, March 2013, Accessed November 12, 2016. [Online]. Available: <http://www.bbc.com/news/health-21829540>
- [179] *Lessons Learned, United Airlines Flight 173*, Federal Aviation Administration (FAA), Washington, DC, Accessed November 12, 2016. [Online]. Available: http://lessonslearned.faa.gov/ll_main.cfm?TabID=1&LLID=42

- [180] Y. Jarraya, M. Debbabi, and J. Bentahar, “On the meaning of SysML activity diagrams,” in *16th Annual International Conference and Workshop on Engineering of Computer Based Systems*. IEEE, 2009, pp. 95–105. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4839236>

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California